

LEARNING JAVA: AN INTRODUCTION TO PROGRAMMING WITH JAVA

The Complete Reference Java Programming for Beginners: A Practical Guide to Learn Java in 10 Days or Less, with Hands-on Projects.

NEOS THANH



LEARNING JAVA: AN INTRODUCTION TO PROGRAMMING WITH JAVA

The Complete Reference Java Programming for Beginners: A Practical Guide to Learn Java in 10 Days or Less, with Hands-on Projects.

NEOS THANH

MENU

<u>Lesson 1: An Overview of Java</u>	14
<u>I. A Little Walk Through The History Of Java</u>	14
<u>II. Java Key Features</u>	14
<u>III. Why Should You Choose Java?</u>	15
<u>1. Simple (Syntax)</u>	Syntax.....	15
<u>2. Fully Object-Oriented (OOP)</u>	16
<u>3. Independence With Hardware And Operating System Platform</u>	16
<u>4. Is A Powerful Language</u>	16
<u>IV. Java Software Development Environment</u>	16
<u>1. Operating System</u>	16
<u>2. Java Development Kit (JDK)</u>	16
<u>3. The Compiler Tool</u>	17
<u>Lesson 2: Installing Development Tools for Java</u>	18
<u>I. Installing the JDK</u>	18
<u>II. Install Eclipse</u>	18
<u>Launch Eclipse</u>	18
<u>Lesson 3: Create Java Project & Get Familiar With Eclipse</u>	18

<u>I. Eclipse</u>	
<u>Overview.....</u>	<u>18</u>
<u>II. Create New Project</u>	
.....	<u>20</u>
<u>III. Creating A New Class.....</u>	<u>23</u>
<u>IV. Hello World!</u>	
.....	<u>26</u>
<u>Lesson 4: Variables and Constants in Java.....</u>	<u>29</u>
<u>I. Variable - Variable</u>	
.....	<u>29</u>
<u>1. Variable Concept.....</u>	<u>29</u>
<u>2. Declare variable</u>	
.....	<u>29</u>
<u>3. Practicing Variable Declaration</u>	
.....	<u>30</u>
<u>4. Name the Variable</u>	
.....	<u>30</u>
<u>5. Variable Data Type.....</u>	<u>33</u>
<u>II.</u>	
<u>Constant.....</u>	
<u>38</u>	
<u>Lesson 5:</u>	
<u>Operator.....</u>	
<u>40</u>	
<u>I. Expression</u>	
.....	<u>40</u>
<u>II. Assignment Operator</u>	
.....	<u>41</u>
<u>III. Arithmetic Operator</u>	
.....	<u>42</u>
<u>IV. Unary Operator</u>	
.....	<u>44</u>
<u>V. Relational Operator</u>	
.....	<u>46</u>
<u>VI. Conditional Operators</u>	

.....	48
<u>VII. Bitwise Operator</u>	
.....	50
.....	
<u>Exercise Number 8</u>	
.....	51
<u>Lesson 6: Type Casting & Comment Source Code</u>	
.....	53
<u>I. Casting</u>	
<u>Concept</u>	53
<u>1. Types Of Casting</u>	53
<u>2. Example Implicit Casting</u>	54
<u>3. Example Explicit Pressing</u>	56
<u>II. Comment Source Code</u>	
.....	57
<u>1. Comment Method</u>	
.....	58
<u>2. Example Comment</u>	58
<u>Lesson 7: Import / Export on Console</u>	
<u>I. Console Concept</u>	60
<u>II. Enter On Consolse</u>	
.....	61
<u>1. Code Faster</u>	
.....	61
<u>a. Crl + Space</u>	62
<u>b. Using Hints From Eclipse</u>	
.....	63
<u>c. Manually Import</u>	
.....	64
<u>2. String Type Data Import Example</u>	
.....	64

<u>3. Data Input Example Type int</u>	65
<u>4. Float Data Import Example</u>	66
<u>III. Exporting on Console</u>	66
<u>Lesson 8: Conditional Statements</u>	68
<u>I. Flow Control Command Concept</u>	68
<u>II. Block Concept</u>	68
<u>III. Scope of a variable</u>	70
<u>IV. The if statement</u>	73
<u>1. if</u>	73
<u>2. if else</u>	74
<u>3. if else if</u>	74
<u>4. ?:</u>	75
<u>V. Switch case statement</u>	76
<u>Lesson 9: Loop Statement</u>	79
<u>I. Repeating Concept</u>	79
<u>II. Repeat Statements</u>	79
<u>1. while</u>	79
<u>2. do while</u>	81

<u>3. for</u>	83
<u>Exercise Number 5</u>	85
<u>Lesson 10: Break And Continue In The Loops</u>	87
<u>I. break - Stop Statement</u>	87
<u>II. continue - Skip Statement</u>	89
<u>Lesson 11:</u>		
<u>Array</u>	
<u>91</u>		
<u>I. Array Concept</u>	91
<u>II. Why Use Arrays?</u>	92
<u>III. Array Usage</u>	92
<u>1. Array Declaration</u>	93
<u>2. Allocate Memory For Arrays</u>	93
<u>3. Array Initialization</u>	94
a. The first way	94
b. The Second Way	94
c. Different ways	95
<u>4. Access Arrays</u>	96
<u>IV. Some Practice With Arrays</u>	97
<u>Lesson 12: Array (Continue)</u>	100

<u>I. foreach</u>	100
<u>1. Foreach Concept Through Questions</u>	100
<u>2. How to Use foreach</u>	100
<u>3. Practice #</u>	1.....	101
<u>4. When Should You Not Use foreach?</u>	101
<u>II. Two-dimensional array</u>	102
<u>1. Declare Two-Dimensional Arrays</u>	102
<u>2. Allocate Memory For Two-Dimensional Arrays</u>	103
<u>3. Creating Two-Dimensional Arrays</u>	103
a. The first way	103	
b. The Second Way	104	
c. Different ways.....	105	
<u>4. Practice #</u>	2.....	106
<u>III. Is There Any More Array Types? !!</u>	110
<u>1. Three-Dimensional Arrays</u>	110
<u>2. Serrated Plate</u>	111
<u>Lesson 13: String</u>	112
<u>I. Chain Concept</u>	112
<u>II. Declare And Initialize A Chain</u>	112
<u>III. Some Useful Chain Methods</u>	

<u>1. String</u>	
<u>Comparison</u> 113
<u>a. equals ()</u> 113
<u>b. Practice # 1</u> 113
<u>c. equalsIgnoreCase ()</u> 114
<u>d. Practice # 2</u> 114
<u>e. Comparison With Operator ==</u> 114
<u>f. compareTo ()</u> 114
<u>g. Exercise Number 3</u> 115
<u>2. Chain</u>	
<u>Joins</u> 115
<u>a. Matching String With Operator +</u> 115
<u>b. concat ()</u> 116
<u>c. Exercise Number 4</u> 116
<u>3. Child Chain Extract</u>	
<u>.....</u>	116
<u>a. subString (int startIndex)</u> 116
<u>b. Exercise Number 5</u> 116
<u>c. subString (int startIndex, int endIndex)</u> 116
<u>d. Exercise Number 6</u> 117
<u>4. Convert In Capital - Normal Print</u>	
<u>.....</u>	117
<u>a. toUpperCase ()</u> 117

<u>b. Practice # 7</u>	<u>117</u>
<u>c. toLowerCase ().....</u>	<u>117</u>
<u>5. Some Other Common Chain Methods.....</u>	<u>117</u>
<u>a. trim ()</u>	<u>118</u>
<u>b. Exercise Number 8</u>	<u>118</u>
<u>c. startsWith () And endsWith ().....</u>	<u>118</u>
<u>d. Exercise No. 9</u>	<u>118</u>
<u>e. charAt ()</u>	<u>118</u>
<u>f. Exercise Number 10</u>	<u>118</u>
<u>g. replace ()</u>	<u>119</u>
<u>h. Exercise No. 11</u>	<u>119</u>
<u>i. indexOf ()</u>	<u>119</u>
<u>j. Exercise No. 12</u>	<u>119</u>
<u>Lesson 14: String Buffer and String Builder</u>	<u>120</u>
<u>I. Why Know StringBuffer And StringBuilder?.....</u>	<u>120</u>
<u>II. What is the difference between StringBuffer and StringBuilder?</u>	<u>121</u>
<u>III. How To Use StringBuffer And StringBuilder.....</u>	<u>121</u>
<u>1. Declaration and Initialization</u>	<u>121</u>
<u>2. Chain Concatenation - append ()</u>	<u>122</u>
<u>3. Insert String - insert</u>	

<u>Q.....</u>	<u>122</u>
<u>4. Replace - replace ()</u>	<u>123</u>
<u>5. Delete String - delete ()</u>	<u>123</u>
<u>6. Reverse String - reverse ()</u>	<u>124</u>
<u>7. Checking the Buffer Capacity - capacity ()</u>	<u>124</u>
<u>Lesson 15: An Overview of Object Oriented Programming</u>	
<u>.....</u>	<u>126</u>
<u>I. Ways of Thinking in</u>	
<u>Programming.....</u>	<u>126</u>
<u>II. Building Applications In the "Old" Direction</u>	
<u>.....</u>	<u>127</u>
<u>III. Object Oriented Application</u>	
<u>Building.....</u>	<u>130</u>
<u>IV. So in short, What Is Object-</u>	
<u>Oriented?.....</u>	<u>133</u>
<u>Lesson 16: Objects &</u>	
<u>Classes.....</u>	<u>135</u>
<u>I. Concept of Objects (Object)</u>	
<u>.....</u>	<u>135</u>
<u>1. Subject's Status</u>	
<u>.....</u>	<u>135</u>
<u>2. Subject's</u>	
<u>Behavior.....</u>	<u>135</u>
<u>II. Class</u>	
<u>Concept.....</u>	<u>136</u>
<u>1. Example 1: Baking</u>	
<u>.....</u>	<u>136</u>
<u>2. Example 2: Designing Car</u>	
<u>Models.....</u>	<u>136</u>
<u>3. Example 3: Doing Student Management Software.....</u>	
<u>137</u>	
<u>III. Class Declaration</u>	
<u>.....</u>	<u>139</u>
<u>1. Class</u>	
<u>Visualization.....</u>	

<u>139</u>	
<u>2. Class Declaration Syntax</u> 139
<u>IV. Declaration And Object Creation</u> 141
<u>Lesson 17: Constructor</u> 143
<u>I. Constructor</u>	
<u>concept</u> 143
<u>II. Constructor</u>	
<u>Declaration</u> 143
<u>1. Creating Constructors For The Circle</u>	
<u>Class</u> 144
<u>2. Practice Initializing Values Through Constructor</u> 147
<u>III. Declaring Objects Through</u>	
<u>Constructor</u> 149
<u>Lesson 18: Getting Started With Inheritance</u> 153
<u>I. Familiar With Inheritance</u>	
 153
<u>1. What Is</u>	
<u>Inheritance?</u> 153
<u>2. Why Must Inherit?</u> 154
<u>II. Inheritance In</u>	
<u>Java</u> 154
<u>III. Familiarize With Class Diagrams</u> 155
<u>Lesson 19: this and super keywords</u> 168
<u>I. This</u>	
<u>Keyword</u>
<u>168</u>	
<u>1. Use this when accessing properties and methods in class</u>
<u>168</u>	
<u>2. Use this When Calling Another Constructor Inside Class</u> 169
<u>3. Use this as the Parameter to Pass to Another Method or Constructor</u>

<u>171</u>	
<u>4. Use this As An Expression of Returned Results</u>	
.....	<u>171</u>
<u>II. Keyword</u>	
<u>super</u>	<u>173</u>
<u>1. Using super when accessing properties and methods of the nearest parent class</u>	
<u>173</u>	
<u>2. Using super When Calling A Constructor Of The Nearest Parent Class</u>	
.....	<u>174</u>
<u>Lesson 20: Getting Started With Inheritance</u>	
.....	<u>176</u>
<u>I. Familiar With Inheritance</u>	
.....	<u>176</u>
<u>1. What Is Inheritance?</u>	<u>176</u>
<u>2. Why Must Inherit?</u>	
.....	<u>177</u>
<u>II. Inheritance In Java</u>	<u>177</u>
<u>III. Familiarize With Class Diagrams</u>	
.....	<u>178</u>
<u>Lesson 21: Overriding In Inheritance</u>	
.....	<u>191</u>
<u>I. What Is Overriding?</u>	<u>191</u>
<u>II. What is Overriding?</u>	<u>191</u>
<u>Lesson 22: Object class</u>	
.....	<u>197</u>
<u>I. What is Object class?</u>	
.....	<u>197</u>
<u>II. What Does the Object Class Do?</u>	
.....	<u>197</u>
<u>III. Methods The Object Class Provides</u>	
.....	<u>198</u>
<u>1. public final Class getClass()</u>	
.....	<u>198</u>
<u>2. int hashCode ()</u>	

.....	199
<u>3. boolean equals (Object obj)</u>	199
<u>4. Object clone()</u>	200
<u>5. String toString()</u>	201
<u>6. void finalize()</u>	201
<u>Lesson 23: Acessibility (Access Modifier)</u>	203
<u>I. What Is Access Modifier?</u>	203
<u>II. What is the Definition of Accessibility?</u>	203
<u>III. Meaning Of Accessibility</u>	204
<u>1. Private Accessibility</u>	205
<u>2. Accessibility default</u>	207
<u>3. Public Access</u>	211
<u>Lesson 24: Final keyword.</u>	212
<u>I. Final Variables & Final Properties</u>	212
<u>II. Final Method</u>	214
<u>III. Final class</u>	216
<u>Lesson 25: Getter and Setter methods</u>	217
<u>I. What is Getter and Setter?</u>	217
<u>II. How To Organize Getter And Setter Practices?</u>	217
<u>III. Use Eclipse to declare Getter and Setter</u>	220

<u>IV. Getter, Setter And Customizations</u>	
.....	221
<u>Lesson 26:</u>	
<u>Static.....</u>	
<u>224</u>	
<u>I. Static concept</u>	
.....	224
<u>II. Uses Of Static</u>	
<u>Keywords.....</u>	
<u>226</u>	
<u>III. Static Property Declaration</u>	
.....	226
<u>1. Example Counting Geometry Declared In A Project</u>	
.....	226
<u>2. Example Declare Static Configuration Values For The Application</u>	
.....	228
<u>IV. Static Method Declaration</u>	
.....	229
<u>Lesson 27:</u>	
<u>Overloading.....</u>	
<u>233</u>	
<u>I. What is Overloading?</u>	
.....	233
<u>II. What Does Overloading Methods</u>	
<u>Do?.....</u>	
<u>234</u>	
<u>Lesson 28:</u>	
<u>Polymorphism.....</u>	
<u>244</u>	
<u>I. What Is Polymorphism?</u>	
.....	244
<u>II. How to Use Polymorphism?</u>	
.....	245
<u>Lesson 29: Casting (Again) in OOP</u>	
.....	256
<u>I. Implicit Castring</u>	
.....	256
<u>1. Implicit casting among same class</u>	
<u>object.....</u>	
<u>256</u>	
<u>2. Squeeze The Class From Child Class To Father's</u>	
<u>Class.....</u>	
<u>258</u>	

<u>II. Explicit Casting</u>	260
<u>Lesson 30: Abstraction in Java</u>	263
<u>I. What Is Abstraction?</u>	263
<u>II. Abstract Class Declaration Like?</u>	263
<u>III. Why Be Abstract?</u>	264
<u>1. Practicing in Building Salary Application for Employees</u>	265
<u>a. Describe Program Requirements</u>	265
<u>b. App Enhancing By Building Abstract Employee Class</u>	265
<u>c. Class diagram</u>	266
<u>d. Building Classes</u>	267
<u>2. Experience Some Abstract Layers Of Java Platform.</u>	274
<u>Lesson 31: Nested Classes</u>	277
<u>I. What Is Cage Class?</u>	277
<u>II. When Should I Use The Cage & Its Use?</u>	282
<u>Lesson 32: Anonymous Class</u>	285
<u>I. What Is Anonymous Class?</u>	285
<u>II. Identity Class</u>	285
<u>Anonymous</u>	285
<u>III. When Should Anonymous Classes Be Used?</u>	286
<u>IV. Class Anonymous</u>	290
<u>1. Anonymous Classes Created Through Inheritance From Another Class</u>	

.....	290
<u>2. Anonymous Classes Created Through Implementation From Another Interface</u>	<u>290</u>
<u>3. Anonymous Class Is Used As A Pass Parameter</u>	
.....	291
<u>V. Anonymous Class Characteristics</u>	
.....	291
<u>Lesson 33: Wrapper Class</u>	<u>293</u>
<u>I. Familiar With Wrapper Class</u>	
.....	293
<u>II. Why Use Wrapper Class?</u>	
.....	294
<u>III. Switching Between Primitive and Wrapper</u>	
.....	294
1. Convert Original Style to Wrapper Style	294
2. Convert Wrapper to Original Style	295
<u>IV. Helpful Methods Of Wrapper Class</u>	
.....	296
1. parseXxx ()	297
2. toString ()	297
3. xxxValue ()	297
4. compareTo ()	298
5. compare ()	299
6. equals ()	299
<u>V. Familiar With Wrapper Class</u>	
.....	301
<u>VI. Why Use Wrapper Class?</u>	
.....	302
<u>VII. Switching Between Primitive and</u>	

<u>Wrapper.....</u>	302
<u>1. Convert Original Style to Wrapper</u>	
<u>Style.....</u>	302
<u>2. Convert Wrapper to Original</u>	
<u>Style.....</u>	303
<u>VIII. Helpful Methods Of Wrapper Class</u>	
<u>.....</u>	304
<u>1. parseXxx ()</u>	
<u>.....</u>	305
<u>2. toString</u>	
<u>().....</u>	305
<u>3. xxxValue ()</u>	
<u>.....</u>	305
<u>4. compareTo</u>	
<u>().....</u>	306
<u>5. compare ()</u>	
<u>.....</u>	307
<u>6. equals ()</u>	
<u>.....</u>	307
<u>Lesson 34: Exception (Part 1)</u>	
<u>.....</u>	309
<u>I. Exception</u>	
<u>Concept.....</u>	309
<u>II. Exception</u>	
<u>Classification.....</u>	310
<u>1. Checked</u>	
<u>Exception.....</u>	310
<u>2. Unchecked Exception</u>	
<u>.....</u>	311
<u>3.</u>	
<u>Error.....</u>	
<u>312</u>	
<u>III. Hierarchical Tree Of Exception Classes In Java</u>	
<u>.....</u>	312
<u>Lesson 35: Exception (Part 2)</u>	
<u>.....</u>	315
<u>I. Be Familiar With Try</u>	
<u>Catch.....</u>	315
<u>1. Practice Building Try Catch</u>	

.....	315
<u>2. Practice Verifying How Try Catch Works</u>	
.....	317
<u>II. Some Rules With Try</u>	
<u>Catch.....</u>	318
<u>1. The Stream Don't Always Get In The</u>	
<u>Catch.....</u>	318
<u>2. But If You Defined Try, You Must Define</u>	
<u>Catch.....</u>	318
<u>3. Can Catch With Exception Class?</u>	
.....	318
<u>III. Try With Many</u>	
<u>Catch.....</u>	319
<u>1. Practice Building Try With Many Catch</u>	
.....	319
<u>Lesson 36: Exception (Part 3)</u>	
.....	326
<u>I. Try Catch With Finally</u>	
.....	326
<u>1. Just There is Try - Finally (Without Catch) Also</u>	
<u>Okay.....</u>	327
<u>2. You May Not Need To Finally, But If There Is Finally in The Code, Finally</u>	
<u>Will Always Be Called</u>	
<u>Last.....</u>	327
<u>3. In Finally Can Still Have Try Catch In</u>	
<u>It.....</u>	328
<u>4. Practice Building Try Catch With Finally</u>	
.....	328
<u>II. Try Catch With</u>	
<u>Resource.....</u>	331
<u>1. Not Any Resource Can Be Used In Try Catch With Resource</u>	
.....	332
<u>2. During Closing System Resources, If An Error Occurs There Will be No</u>	
<u>Exception Of This Process</u>	
.....	332
<u>3. Multiple Resources Can Be Generated Inside Try Block</u>	

.....	332
<u>4. Practice Building Try Catch With Resource</u>	
.....	332
<u>5. Practice Building Try Catch With Many Resources</u>	
.....	333
<u>III. Some Helpful Methods Of The Exception Class</u>	
<u>1. getMessage</u>	
()	334
<u>2. toString</u>	
()	334
<u>3. printStackTrace</u>	
()	335
<u>Lesson 37: Exception (Part 4)</u>	
.....	337
<u>I. Throw - Throw An Exception</u>	
.....	337
<u>II. Throws - Throwing Exception For Elsewhere Handled</u>	
.....	340
<u>III. Create Your Own Exception</u>	
.....	344
<u>Lesson 38: Thread Part1 - Thread Concept</u>	
.....	347
<u>I. Thread Concept, Or Multithread</u>	
.....	347
<u>II. Distinguishing Related Concepts</u>	
.....	348
<u>III. When to Use Thread</u>	
.....	349
<u>Lesson 39: Thread (Part 2)</u>	
.....	354
<u>I. Create A Thread</u>	
.....	354
<u>1. Method 1 - Inheriting From Thread Class</u>	
.....	354
<u>2. Practice Creating A Thread By Inheriting Thread Class</u>	
.....	355
<u>3. Method 2 - Implement From Interface Runnable</u>	
.....	356

<u>4. Practice Creating A Thread By Implementing From Interface Runnable</u>	
.....	357
<u>II. Applying Anonymous Class Knowledge In Creating New A Thread</u>	
.....	357
<u>1. Creating An Anonymous Thread From Inheriting Thread Class</u>	
.....	358
<u>2. Create An Anonymous Thread By Implementing From Interface Runnable</u>	
.....	360
<u>Lesson 40: Thread (Part 3) Life</u>	
<u>Circle</u>	366
<u>I. What Is The Life Cycle Of An Object?</u>	
.....	366
<u>II. Why Should A Person Learn The Life Cycle?</u>	
.....	366
<u>III. Understanding The Life Cycle Of</u>	
<u>Thread</u>	366
<u>1. Thread Lifecycle Illustration Diagram</u>	
.....	366
<u>2. Life Cycle Description Of Thread</u>	
.....	368
<u>IV. Statuses Within A Life Cycle</u>	
.....	370
<u>1. NEW</u>	
.....	370
<u>2.</u>	
<u>RUNNABLE</u>	
371	
<u>3.</u>	
<u>BLOCKED</u>	
372	
<u>4. WAITING</u>	
.....	374
<u>5.</u>	
<u>TIMED_WAITING</u>	
377	
<u>6. TERMINATED</u>	
.....	379
<u>Lesson 41: Synchronization (Part</u>	
<u>1)</u>	381

<u>I. What is Synchronization?</u>	381
<u>II. When Will Synchronize Be Used?</u>	382
<u>III. Methods of Synchronization</u>	385
<u>Lesson 42: Synchronization Part 2</u>	
.....	387
<u>I. What is Mutual Exclusive Sync?</u>	387
<u>II. How is Mutual Exclusive Sync?</u>	389
<u>III. Keyword synchronized</u>	390
<u>1. Use synchronized For Method</u>	390
<u>2. Use synchronized For Method Inside Block</u>	393
<u>Lesson 43: Synchronization Part 3 Sync Cooperation</u>	
.....	397
<u>I. What is Sync Cooperation?</u>	397
<u>II. How to Sync Cooperation?</u>	397
<u>1. wait</u>	
<u>()</u>	398
<u>2. notify ()</u>	398
<u>3. notifyAll ()</u>	398
<u>III. Practice Solving Withdrawals From Banks</u>	
.....	398
<u>Lesson 44: Deadlock</u>	
.....	406
<u>I. What is Deadlock?</u>	406
<u>1. Understanding Deadlock Through Realistic Examples</u>	406
<u>2. Deadlock In Programming</u>	

.....	407
<u>II. Application Construction Practices Causing Deadlock</u>	
.....	407
<u>III. When Did Deadlock Appear?</u>	
.....	411
<u>IV. How To Avoid Deadlock And How To Handle It If Meeting Deadlock?</u>	
.....	411

Lesson 1: An Overview of Java



I. A Little Walk Through The History Of Java

Let's begin to familiarize ourselves with this curriculum through a brief understanding of the Java language.

Java was created by *James Gosling and his team at Sun Microsystem* in 1991 (*Oracle later acquired Sun Microsystem in 2010*). Initially this language was

called *Oak* (oak tree) because outside the company at that time there were many trees planted (other documents say so, I just

copy). Oak was officially renamed Java in 1995, probably because the oak trees were cut down.

II. Java Key Features

Java is an *Object Oriented Programming Language (OOP)* . So when programming with this language you will have to work with classes. Java's syntax *is heavily borrowed from C / C ++* but has simpler object-oriented features and less low-level processing features, so it will be easier to approach Java than C / C ++, plus if you already have a foundation on C / C ++ will surely accept and approach Java more easily.



The famous slogan of Java as you probably knew is "*Write Once, Run Anywhere*" . Writing here is writing code, while executing the application. This means, software language can be run on any different platforms (platforms). To do this, Java introduces the concept of the *JVM (Java Virtual Machine)*, when you compile a program, instead of the source code will be translated directly to machine code like many other languages, then running means written in Java

with Java code That source will be translated into *byte code* , this byte code will be distributed by you to different devices, the JVM built in those devices will further translate this byte code into machine code for you. This compilation can be described with the following diagram.



III. Why Should You Choose Java?

1. Simple (Syntax) Syntax

As I said above, since Java is inherited from C / C ++, it will still retain the simplicity of syntax compared to what C / C ++ has achieved.

On the other hand, Java also reduces the "*headache*" concepts that C / C ++ is having, making the language simpler and easier to use. Some of these reductions can be mentioned such as: removing *Goto* statements , no longer the concept of *Overload Operator* , and also removing the *Pointer* concept , removing the *Header* file , removing the always *Union* , *Struct* , ...

2. Fully Object-Oriented (OOP)

There are also many opinions surrounding these two words "*completely*" , in fact only *primitive data types* of Java such as *int* , *long* , *float* , ... are not object oriented. In addition to those primitive data types, when exposed to Java, you always have to think and work in an object-oriented manner. So what is object oriented? You will understand because we will start talking about object orientation from the next lessons.

3. Independence With Hardware And Operating System Platform

As mentioned above, Java's motto is "*Write Once, Run Anywhere*" . This made the Java language platform-independent. When programming with Java, you will not have to think about the architecture compatibility of each type of operating system or hardware, it is the JVM that will help you with this.

4. Is A Powerful Language

Java is powerful because it supports many programmers. First, as I mentioned above, Java can run on multiple platforms. Java also has *the Ministry of garbage (Garbage Collection)* that automatically cleans the objects used to release the memory, but with other languages, the programmer must perform a liberating manually. Java also supports *multithreads* very well. And there are many other things we will learn together.

IV. Java Software Development Environment

The Development Environment is an environment where *Software Developers* have the tools most necessary to write a complete application. Since the lesson is related to Java programming, we will focus on understanding what *the Java*

Software Development Environment will consist of.

1. Operating System

No matter what operating system you are programming on: *Windows ,Linux or Mac* . Then you can all install a *Software Development Environment* for Java.

2. Java Development Kit (JDK)

The Java Development Kit (JDK) , which will give you the tools you need to compile, execute, and even have an environment for your Java application to run.

3. The Compiler Tool

The Compiler Tool that I want to talk about here is an *Integrated Development Environment (IDE)* , or in other words a tool for you to write Java code on, this tool can be powerful enough that apart from you. Can code, it also helps check syntax errors when code, helps to associate with the JDK to automatically compile and execute the program.

With Java you have more options, you can use one of the following tools.

Netbeans : It is a free, powerful, open source IDE. However, the tool is a bit outdated today, you probably don't even need to know it.

Eclipse : It is also an open source IDE, its popularity is no less than Netbeans, even when Eclipse was used more commonly. And of course this tool is also free. But it can be said that even though Eclipse was once used by the Java community, with the new IntelliJ tool that I will talk about in the following section, it may make Eclipse as obsolete as Netbeans.

IntelliJ : Although born late, *IntelliJ* quickly won the *hearts* of the Java programming community thanks to its robustness and modern interface. Especially since Android Studio was officially introduced by Google as an official Android application programming tool, but Android Studio was built from IntelliJ, so this IDE suddenly became very hot.

In addition to the three popular IDEs mentioned above, there are many other tools that support Java programming. You can choose for yourself an IDE, but the lessons in your program will agree on Eclipse. You may not have to use the same as yours elf, you can view the code for the lessons, but still compile it on another tool, IntelliJ for example. I will move Java lessons from Eclipse to IntelliJ at a later date, for easy access when you want to learn more lessons from

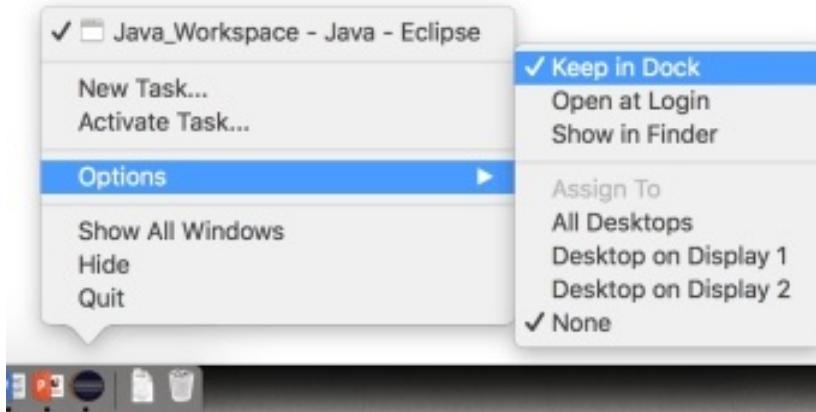
application programming for Android operating system .

Next Lesson You will be instructed to install a Software Development Environment for this Java language .

Lesson 2: Installing Development Tools for Java

Lesson 3: Create Java Project & Get Familiar With Eclipse

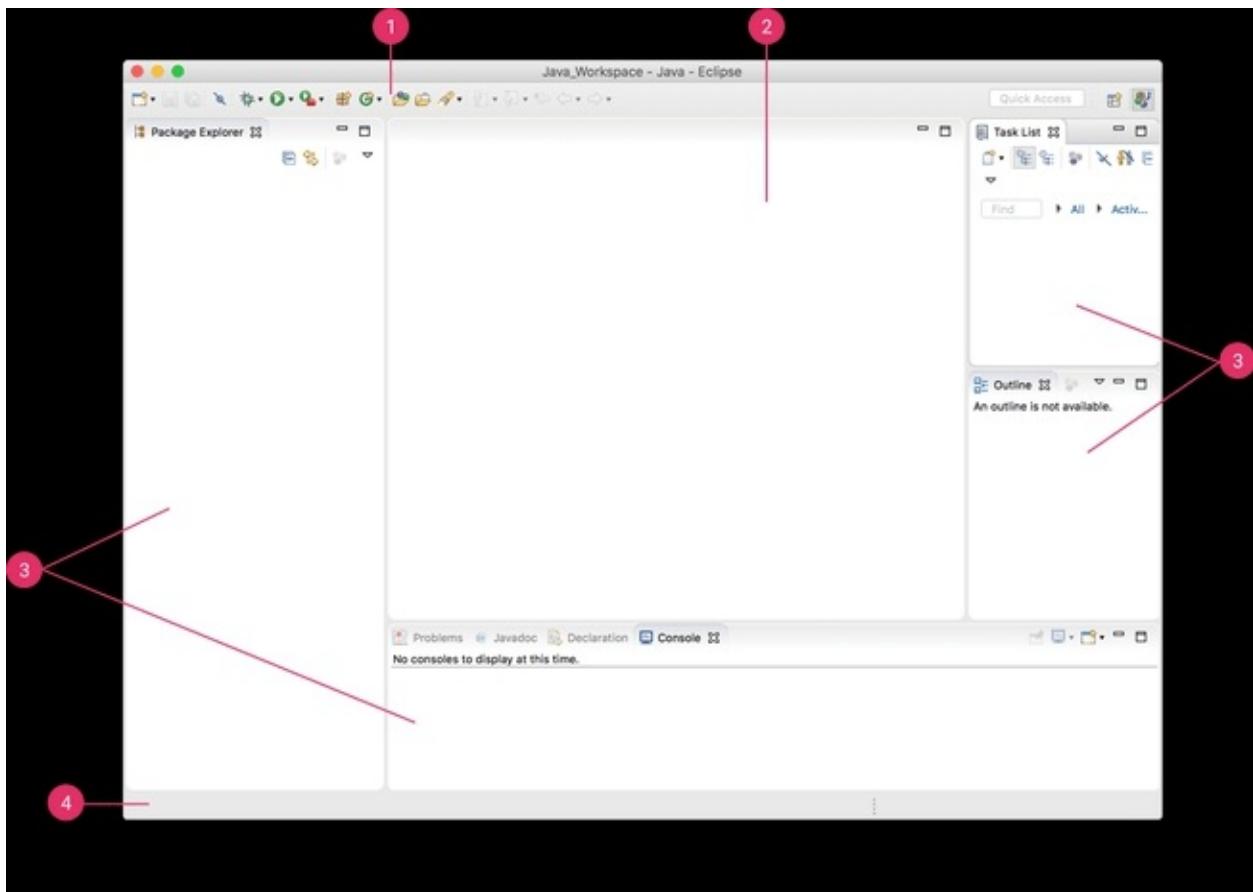
With learning about the language and how to install a Java programming environment from the previous two articles. Today we open up Eclipse to get familiar with the IDE and with your first Java code. When you find Eclipse, then you must create a shortcut right away and always if you are using Windows, but if you are using a Mac, you can "pin" this Eclipse to the dock for later opening, as shown below.



After you have opened Eclipse up, we start on this lesson 3.

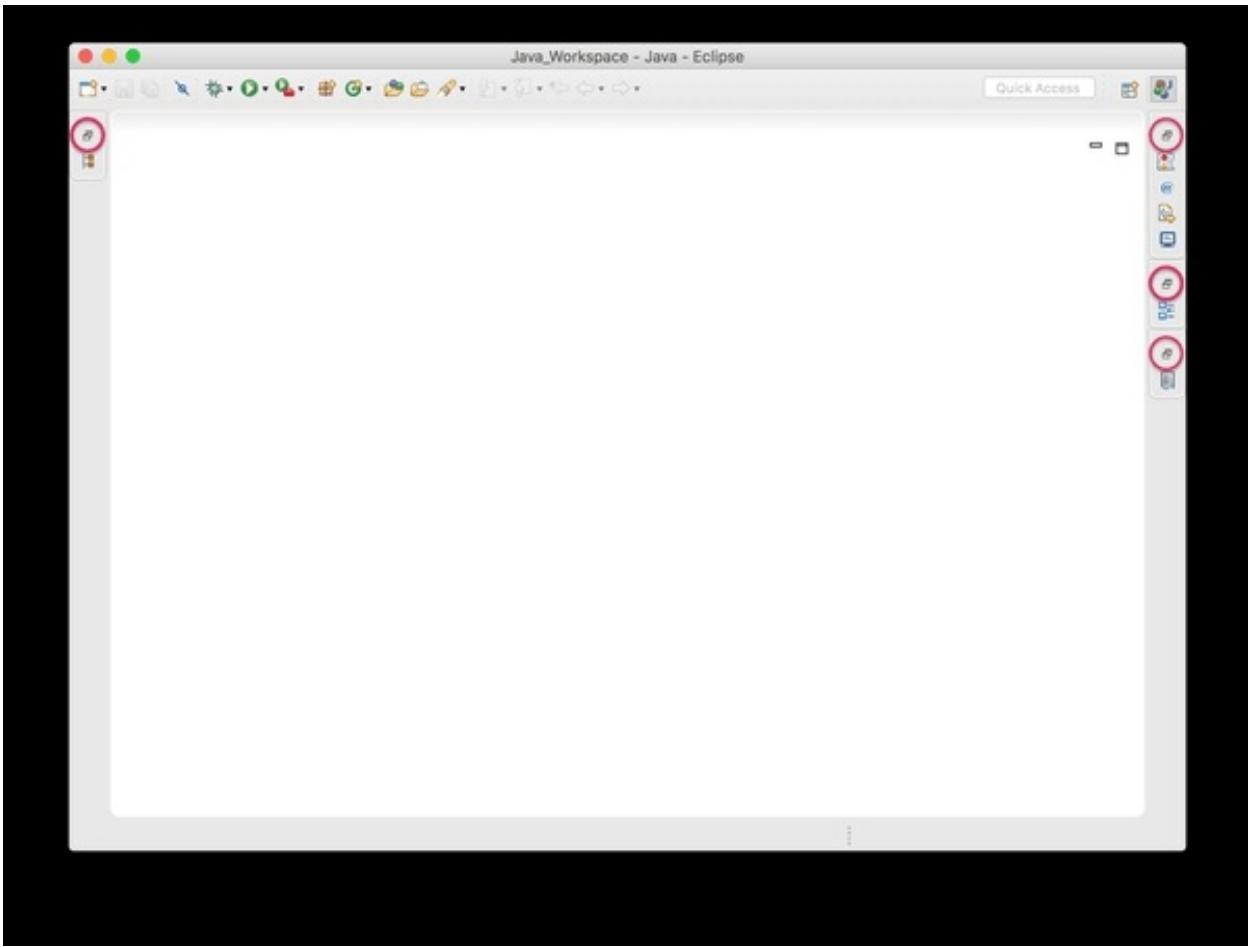
I. Eclipse Overview

The first time when you open Eclipse, you will see the main screen as follows, the children of this screen are numbered for you to easily access.



1. *Toolbar* : This is Eclipse's toolbar, this bar contains control buttons, such as *Create new project , Save project , run / debug project ,...*
2. *Editor* : is where you will code the Java code lines into this window.
3. *View windows* : the monitoring windows, the logs or the project directory tree will be displayed in these windows.
4. *Status bar* : The status bar occasionally shows the status of the application.
For each child window, you can completely click on the mark to shrink them. I think you

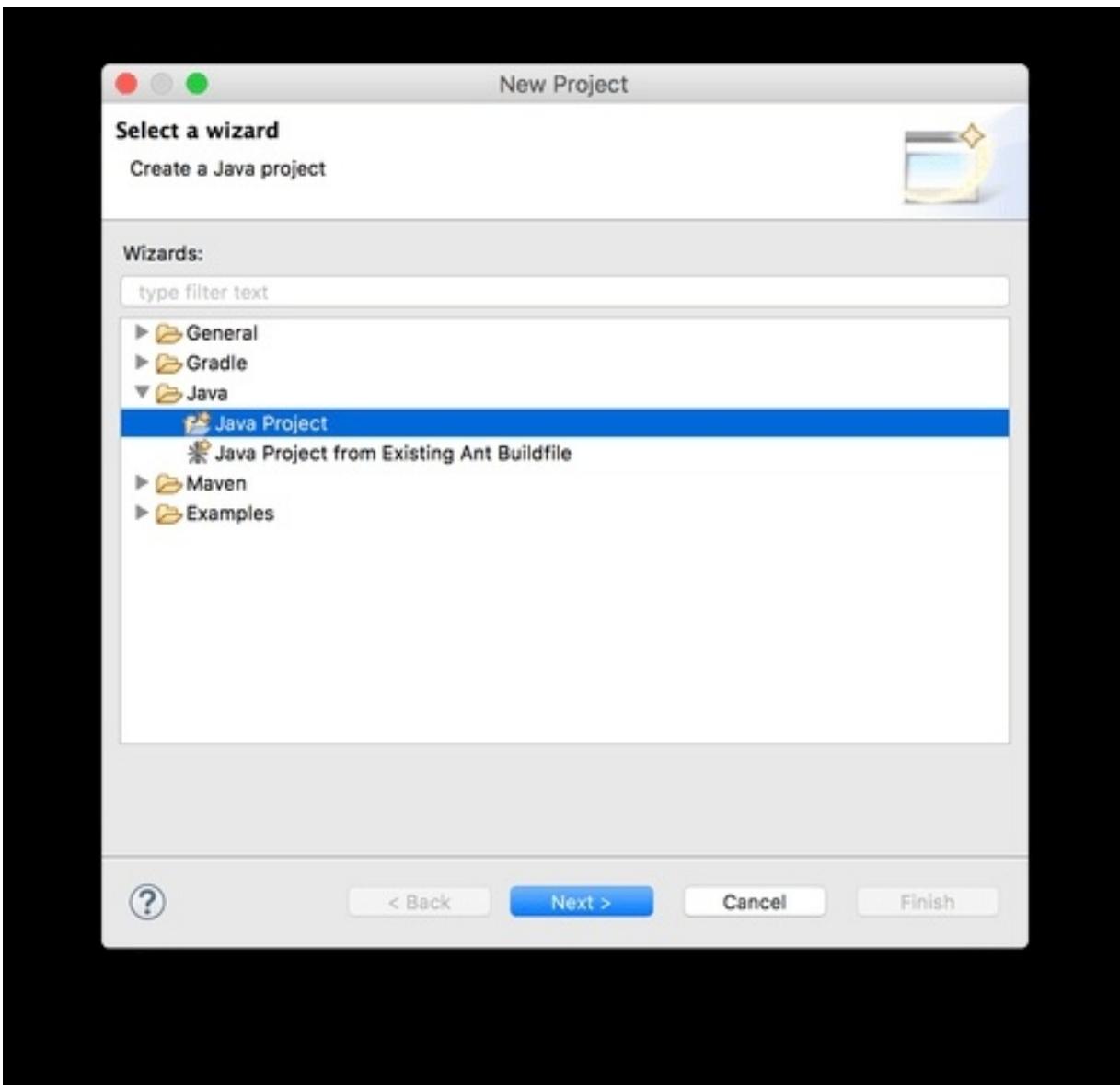
should minimize everything, just leave only the *Editor* window for easy coding, as you can see in the image below. You can rest assured that you can display them the old size when needed by pressing the buttons circled in red again.



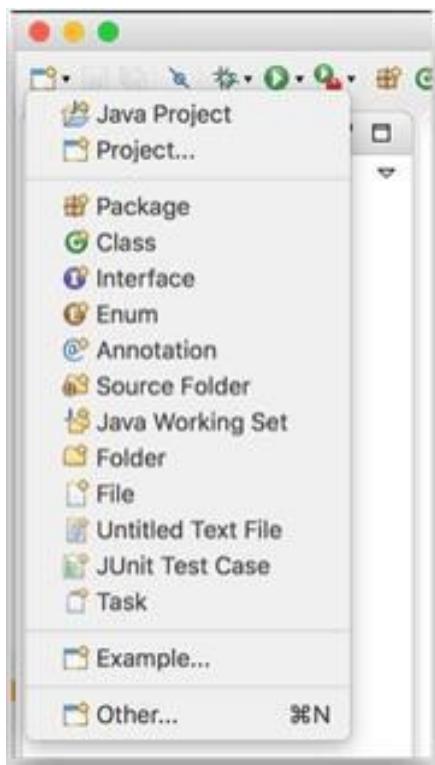
II. Create New Project

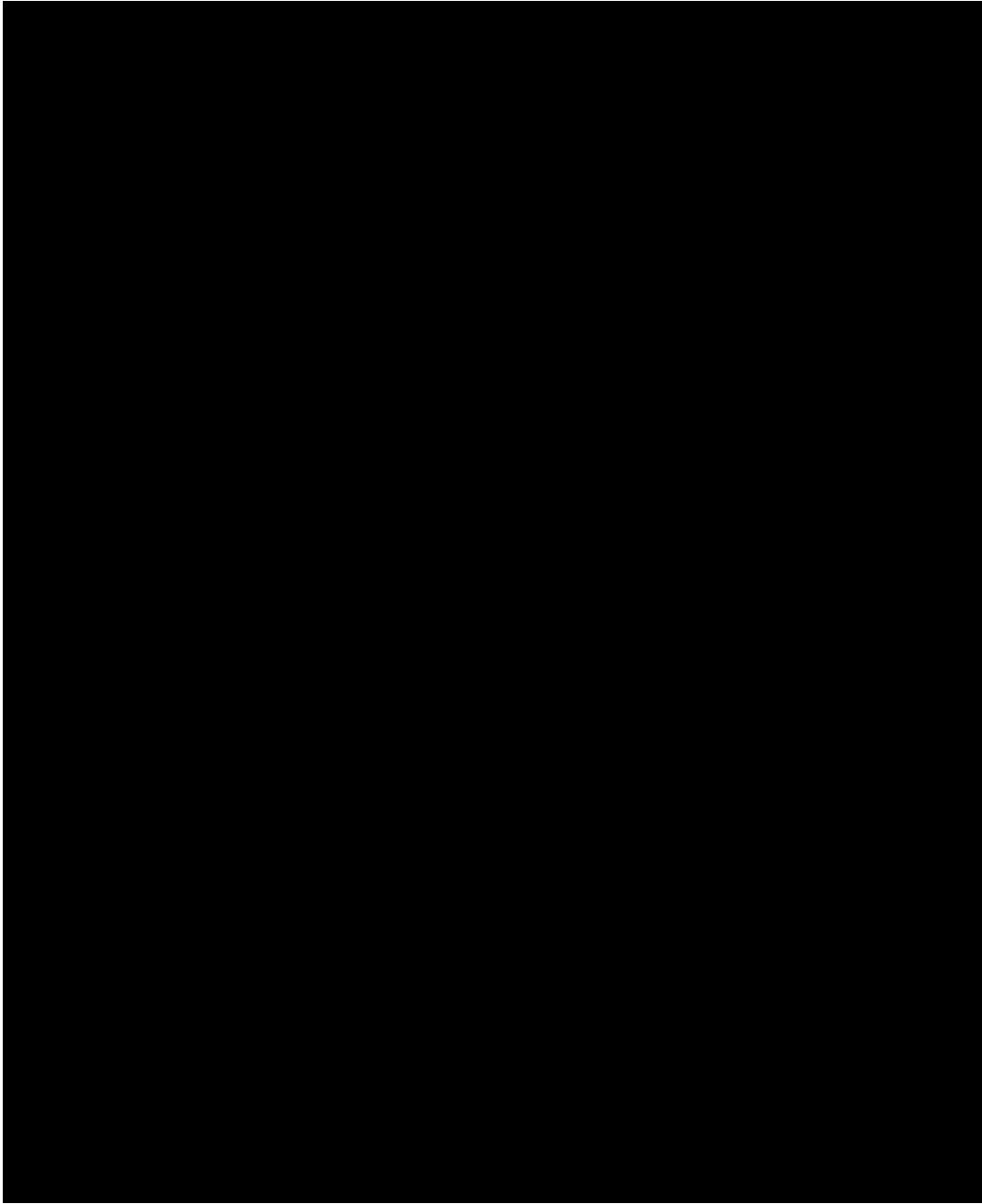
There are following ways to create a new project. Please choose for yourself the way that you like best.

1. Go to *File> New> Java Project* menu.
2. Go to *File> New> Project...* menu. The next window appears, click on the folder named *Java* , in the following components, select *Java Project* and click *Next>* as shown below.



3. If you do not use the menu you can look at *the window-shaped icon with the + sign* in the Toolbar as shown below, but remember to click on the triangle button next to the icon. Then you also have two options like the second method above.





You can choose any of the three ways above, but I'm lazy, I press Command + N (that's for Mac, with Windows it's Ctrl + N) for it quickly.

In the *Project name*, type in the name of the project, I will name this project *HelloWorld* , you can type a space or capitalize as you like. So why set it as "*HelloWorld*" ? The reason your first project has such a name is because it shows that this is your mark with a new programming language, for which the first mark is seen as a greeting to you. to the world. Sounds majestic, actually I'm a little joke, hello world is always used by books or programming guide websites

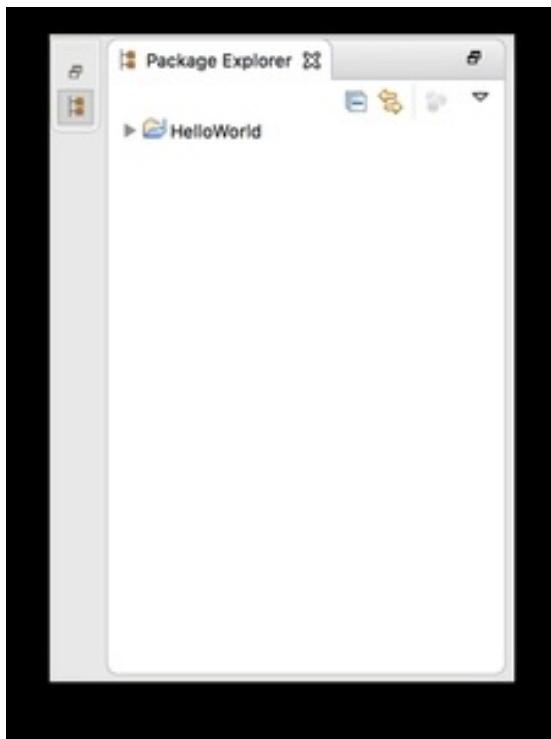
when guiding people in the first lesson, it means to start. for the good things ahead. And my lesson is no exception, hello world! You have the right to name any name in today's post.

Back to the lesson, after you name the project as above, click *Finish*.

III. Creating A New Class

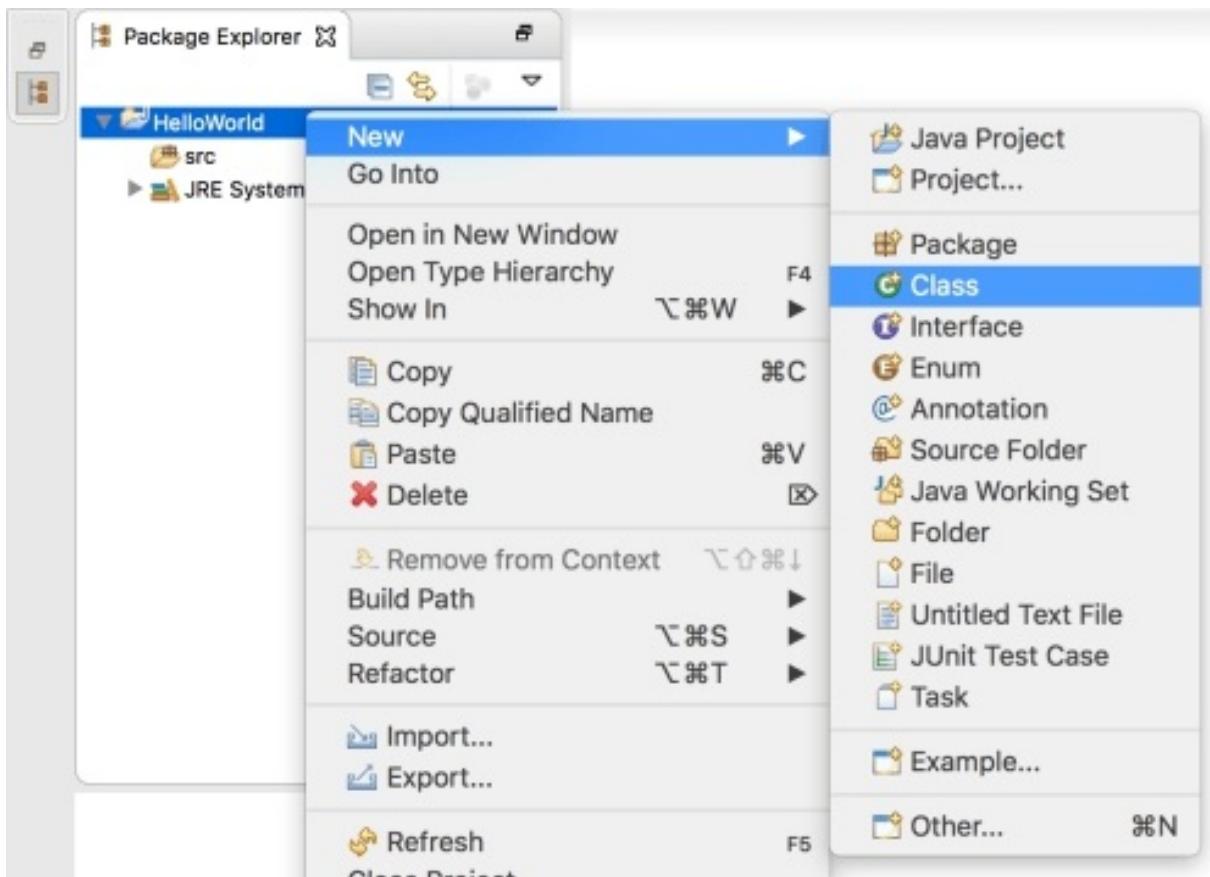
In this step you will create a class. You will wonder *what class is?* As in the previous post, I also said that Java is an object-oriented language (OOP), as soon as you work with Java you are forced to think and work object-oriented whether you are a novice or not. And class is one of the concepts of object oriented. You will learn about the class in the next lessons.

But not just working in an object oriented way is to know about OOP, in these first lessons, you just accept creating a new class. You just need to know that the class is where we will code into, the system will look for the classes to compile the source code into executable code, every line of code outside the class is invalid and the system will report an error immediately.

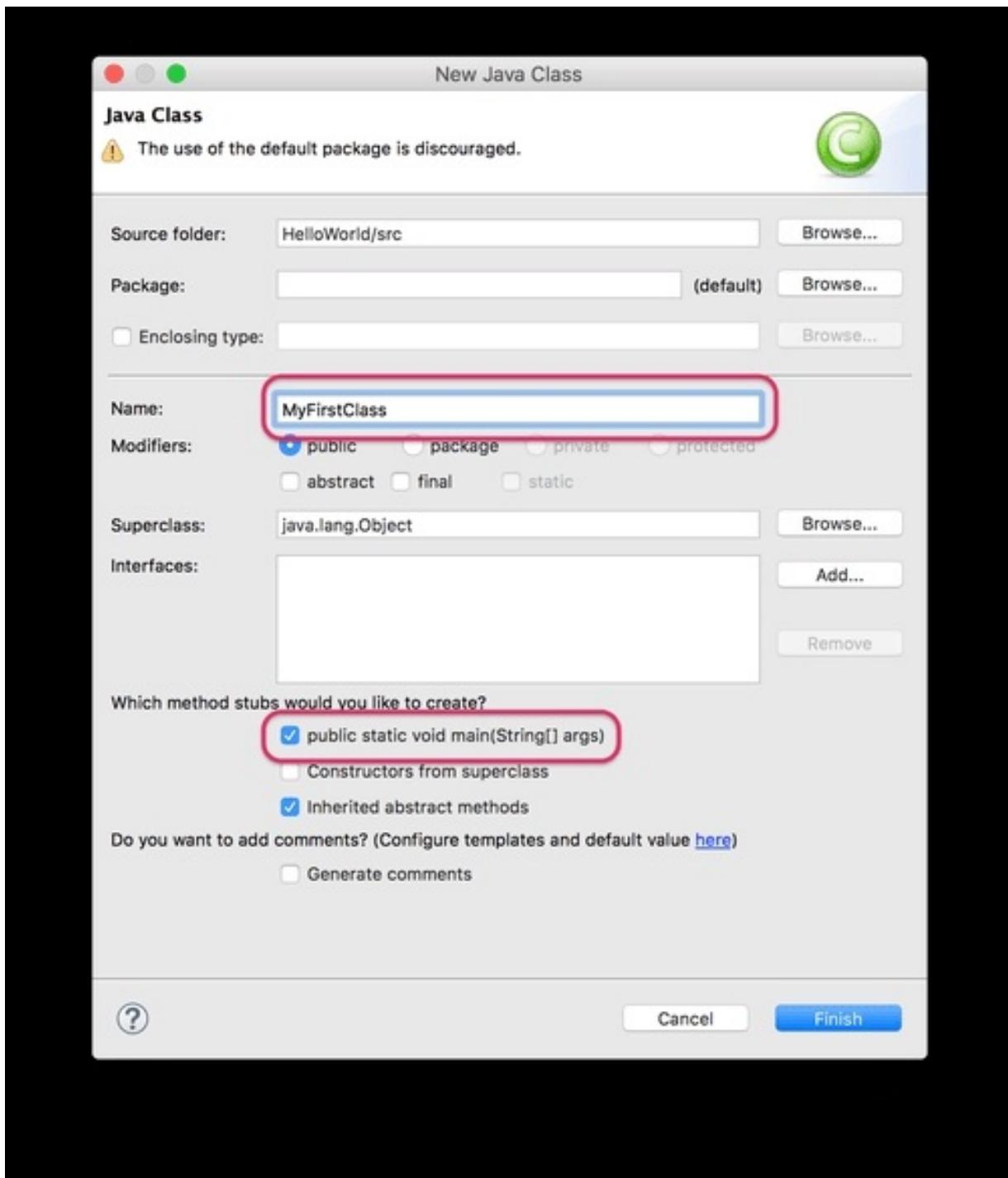


Before creating a new class, make sure that a small window on the left side of

Eclipse is open, this window called *Package Explorer*, which will display all the files and folders in your project in a directory tree, with your project. Just created, *Package Explorer* displays as follows. To create a class, you can choose from the menu *File> New> Class* , or right-click the project in the *Package Explorer* window and choose *New> Class* .



The next window appears, you name the class in the *Name* section, as shown below, I named this class *MyFirstClass* . And remember to check and select *public static void main (String [] args)* , with this option checked, the system will make available for you a *main* method in the newly created class. This *main* method is the method that the system will find first and start executing lines of code from here for you. Without the *main* method , the system will not know where your application starts, and so no lines of code will be executed. You will have a better understanding of the concept of *Methods* as well as the *main* understanding of the *main* method and other methods in the following lessons.



After clicking *Finish*, you will see the *MyFirstClass.java* class appear in the *Package Explorer* window, the content of this class is also open in the *Editor* window, you can also see the *main* method is made available when you check the checkbox. step above. If you do not check the checkbox but want to code yourself, you should code for each word as shown below, you code the wrong word, the system will either error or will not run.

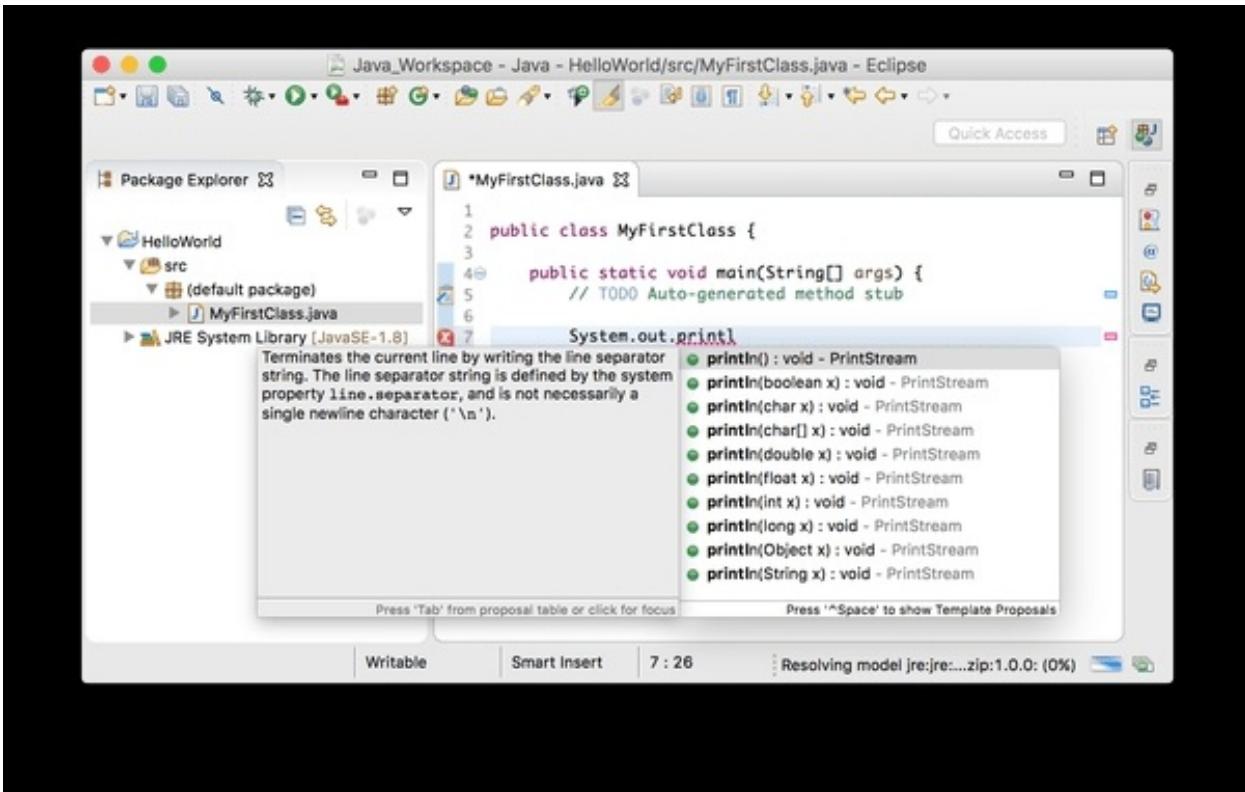
IV. Hello World!

Now it's time to code the first line of Java code. With the class *MyFirstClass.java* opened as above, you code into the *main* method as follows, remember that the end of the statement must have a ";" Please.

```
first 1 public class MyFirstClass {  
2  
3     public static void main(String[] args) {  
4         // TODO Auto-generated method stub  
5  
6         System.out.println("Hello World!");  
7     }  
8  
9 }
```

Maybe you will be lazy because instead of code, you copy / paste the code from this website. Is it possible, is it true you? If it is, you should remove the lines of code just pasted, then code from your own hands. In the following lessons, too, when encountering lines of code or requirements that force you to code, you should not copy, but read the request first, then try the code first. But if you don't know how to code, you can look at the sample code and code it again. Then you try to execute the program to see if the results are correct or not. If it's your own code, and your execution doesn't work for you, either you are wrong, or you are wrong, and you can leave a comment below the lesson to remind yourself. And if the performance of you and I are too good, but the code is different, it's okay, programming is an open mindset, And each of us has a different way of reasoning, as long as we come to the same result. You have a rough understanding of how to learn programming, right.

It is a good note that during the coding process, after you type a period (.) , The system usually suggests methods, then you can press enter (return on Mac) to quickly select the method. Note, or use the arrow keys to select the corresponding other methods.



If you type in the command, you see an x icon in the red circle, this icon appears in the left bar of the Editor as shown above, either the code is faulty somewhere, or you have not finished the code. end the command with ";", even without saving the class,... Any errors that you do not know how to fix, leave a comment below this post, I will help you.

After you are confident in the code, remember to save it by clicking icon on the toolbar. Then click icon to execute the program.

Very quickly, you should see the *Console* window appear with the *Hello World* content you just coded earlier, because the command *System.out.println()* is to *print the log to the console* .

Congratulations, you've just finished coding your first Java program.

Lesson 4: Variables and Constants in Java

With the *HelloWorld* project you created in the previous lesson. Today you start

to learn about the Java language by accessing two concepts *variables* and *constant*, with the lessons you will also be practical to declare some *variables* and *constant* basis. And to be as accessible as possible, with each practice code, I will also give *TRUE* and *FALSE* cases of the code, so that you both understand quickly and understand the lesson. Please start the lesson.

I. Variable - Variable

The first approach to a programming language, not just Java, you must be familiar with the concept of *variables*, English documents call *variables*.

1. Variable Concept

Variables are definitions from you, which help the system create memory areas to hold values in the application. *Each variable has its own* data type, which will tell the system how "*large*" the variable is. The size of the variable will indicate its ability to store values. You will learn about the storage size of each variable in this lesson. In addition, each variable must be given a name by you so that the system can manage and access data in the variable's memory.

2. Declare variable

As the above concept, you probably already have some grasp, the declaration of a variable is essentially that you will name the variable, then set a data type for the variable, and be able to store the original value in the same memory of the variable.

To declare a variable in Java, keep in mind the following syntax:

variable_data type; or variable_data type = value;

I explain a bit about the above syntax.

- *data type* will determine the size of the variable, the types of *data* types and their uses will be discussed in more detail below the lesson.
- *variable name* is... the name of the variable, the way to name the variable will be clearer in the below section.
- When declaring a variable, you can "*assign*" the variable an initial *value*, assigning this value will make the variable get the initial data as soon as it is created. If a variable is created with no value assigned, it will be assigned a default value, which will depend on the *data type* of the variable we will talk about below.

- Finally, after each variable declaration you remember there is a sign ";" to notify the system of the end of the declaration, if there is no ";" the system will report an error immediately.

3. Practicing Variable Declaration

Now, open the *HelloWorld* project just created in the previous lesson, try typing the following variable declaration lines. Just type to get used to it. Pay attention to the highlighted lines. You will have the opportunity to declare variables in the sections below.

```
first  public class MyFirstClass {  
 2  
 3      public static void main(String[] args) {  
 4          // TODO Auto-generated method stub  
 5  
 6          System.out.println("Hello World!");  
 7  
 8          double salary;  
 9          int count = 0;  
ten        boolean done = false;  
 11        long earthPopulation;  
twelfth    }  
 13    }  
 14 }
```

4. Name the Variable

As in the example above, the variable names are named *salary* , *count* , *done* or *earthPopulation* . Naming a variable is simple, no matter how you set it, as long as you can read it (human language), and you find it easy to understand and correct with the function of that variable.

However, the naming of variables is not completely free, there are some "*rules*" and "*rules*" of naming below, if you encounter one of these naming rules, the system will immediately error you..

- Rule 1

The start character of a variable name must be a letter , or an underscore (_), or a dollar character (\$) . Examples for correct variable naming are as follows.

It's int count; correct

int Count; int _count; int \$count;

Also naming variables as follows will fail. Since their first characters are either a number, or a special character is not allowed. If you do not believe, just try typing in Eclipse.

int 5count;

Wrong int 5Count; int #count; int /count;

- Rule 2

There must be no spaces between the characters of the variable . The variable naming example as follows is false.

Wrong int this is count;

If you want the above variable name to be clearly separated from each letter for clarity, you can set the following.

It's int thisIsCount; correct int this_is_count; int This_Is_Count;

At this point, let me speak the margins: in the case of a variable with many words like the above example, it is okay to choose which of the three ways you want to put it, but I recommend you to choose the first *thisIsCount* . This way, the first letter *this* is lowercase, the letters in each subsequent word will capitalize *IsCount* . This is not mandatory, but it becomes a common "rule" for Java programming, if you do this naming way, it will be much easier for you to read other people's Java code since they will write like you. Note that not all letters are capitalized like *ThisIsCount* because it is easy to confuse with naming the class that you will learn in the following lessons, or *THIS_IS_COUNT*. because it is confused with the constant that you will be familiar with in this lesson.

- Rule 3

*Do not contain special characters inside variable names like ! @ # % ^ & * .* The variable naming example as follows is false.

Wrong

int c@unt;

int count#;

int count*count;

- Rule 4

Do not name the variable with the same keyword. . Keywords are keywords that the Java language is dedicated to for several system purposes. Java keywords are listed in the following table.

abstract continue for new switch
assert default goto package synchronized
boolean due if private this
break double implements protected throw
bytes else import public throws
case enum instanceof return transient
catch extends int short try
char final interface static void class finally long strictfp volatile const float native
super while
The variable naming example as follows is false.

Wrong

```
boolean continue = true; long class;  
int final;
```

However, you can name the variable with the keyword inside without getting an error, like in the following example.

It's boolean continue1 = true; correct long classMySchool; int finalTarget;

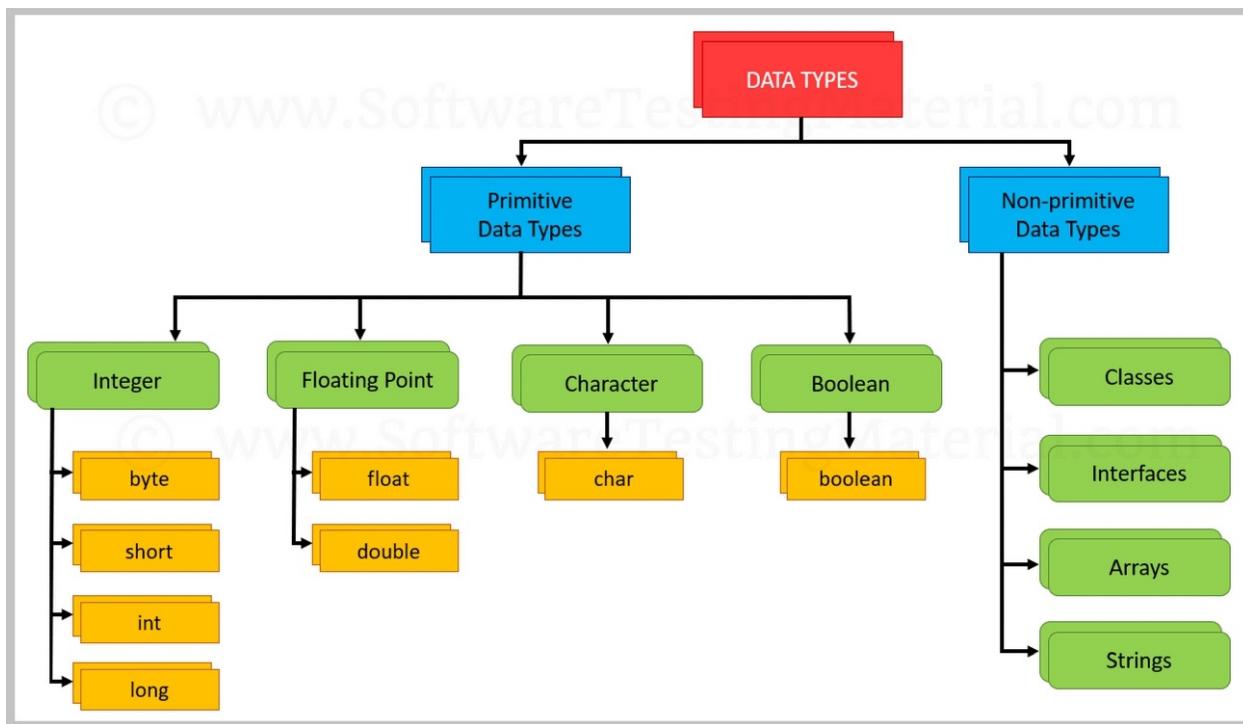
5. Variable Data Type

As mentioned above, each variable must have a *data type* associated with it. The data type will tell the system how “*big*” the variable is , and this will indicate its ability to store values.

It is up to you to choose a data type for the variable. Usually you should base it on the function of the variable, such as a variable of character type or numeric type. Or you can rely on the variable's ability to store values, like an *integer* or a *long* .

Let's get started with 8 “*primitive*” data types of variables. Called *data types primitive* because they are primitive data type and provided availability of Java. In terms of storage, the primitive data type stores the data in itself, using this type is also very simple, and has nothing to do with object orientation. In contrast to the primitive *data type* is the “*extended*” *data type* that I will talk

about in the next lessons when you have grasped the basics of Java. The primitive data types for today's lesson are listed in the following diagram.



Looking at that much, the primitive data types that we need to care about are the yellow cells, including types `int` , `short` , `long` , `byte` , `float` , `double` , `char` , and `boolean` . Other colored boxes are just grouping data types so that they are easier to remember and use

Let's go into each type of data type to know better.

- Integer Type

This type is used to store and calculate whole numbers, including integers with negative values, integers with positive values, and zeros . The reason for there are many types of integers as above, is because depending on the size of the variable that you will declare the corresponding data type, we will look at the table of values of integer data types as follows.

Datatypes^{Storage} Variable Size_{Memory}

bytes 1 byte From –128 to 127

short 2 bytes From –32,768 To 32,767

int 4 bytes From –2,147,483,648 To 2,147,483,647

long

8 bytes

From -9,223,372,036,854,775,808 To 9,223,372,036,854,775,807

As shown in the table above, when you find it very important to choose the size of the type, as with the *byte* type , the system will only allocate a memory area to store exactly 1 *byte* of data, with this memory area you only allowed to use variable sizes *from -128 to 127* only.

For example, you declare the following variables as valid.

It's byte month = 5;correct short salaryUSD = 2000;

The following declaration goes beyond the storage capacity of the variable and so you will get an error immediately.

Wrong byte day = 365;

short salaryVND = 40000000;

So when using data types for variables you must consider the size of their storage. You should not always use the *long* data type for all cases, as this will consume more system memory when the program is started. You should know the limit of the variable to give the right data type. For example, if you use a variable to store the months of the year, *byte* type is sufficient. Or use the variable to store employee salaries in VND, then use the *int* type, but if your program stores all the rich, the salary can be used with

the *long* type.

If a variable is not declared, integer variables will have a default value of 0 (or 0L for *long*).

- Real Number Type This type is used to store and compute real numbers, which are floating point numbers. Just like the integer type, the real number type is also divided into many categories with different magnitudes depending on the purpose, as shown in the following table.

Datatypes Storage Memory Variable Size

float 4 bytes ± 3.40282347E + 38F

double 8 bytes ± 3.40282347E + 38F

With the comfort in memory allocation of real numbers, you don't worry too much about distinguishing when to use *float* or *double* . The following example shows the case of using the real number type.

It's float rating = 3.5f; correct double radius = 34.162;

The reason for the *float* type above must put the letter *f* at the end of the declaration, *float rating = 3.5f*; This is because sometimes we have to emphasize to the system that we are using *float* rather than *double*, if you don't add *f* to the end of the value then the system will get an error. Similarly, you can also declare *double radius = 34.162d*; But because the system already understands this is a *double* number, so you don't have the letter *d* in this case.

If you don't declare a value for a variable, then a variable of type real number will default to *0.0f* for type *float* and *0.0d* for type *double*.

- Char type

The *char* type is used to declare and contain a character. You can assign a character to this type in a pair of single quotes like '*a*' or '*B*' as shown in the following example.

It's correct `char thisChar = 'a';` You must always remember that the '*a*' declared above must be in single quotes, not quotes. The double click is for the *sequence that* will be discussed in the next lesson. And remember that if there is no click is also wrong. The following example incorrectly declares char.

Wrong

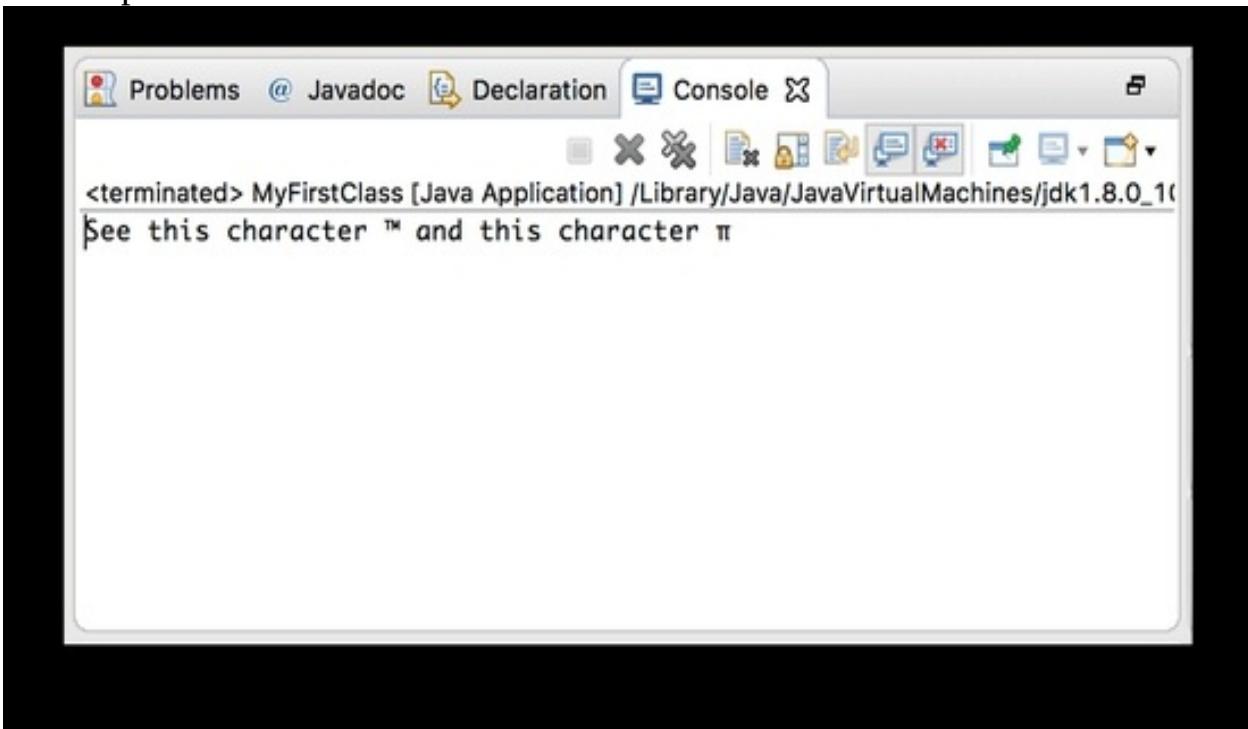
`char thisCharFail1 = "a"; char thisCharFail2 = b; char thisCharFail3 = 'ab';`

In addition to the above declaration, the *char* type is also used in *Unicode*. Unicode codes start with '`\u0000`' and end with '`\uffff`'. Note that using this Unicode encoding, you still have to use single quotes for declaration. The symbol `\u` indicates you are using it with Unicode, not normal characters. With this Unicode-style declaration, you can put some special characters into the program, try the following code practice and print out log, you will see the effectiveness of Unicode.

```
public class MyFirstClass {  
  
    public static void main(String[] args) { // TODO Auto-generated method stub  
        char testUnicode1 = '\u2122';  
        char testUnicode2 = '\u03C0';  
  
        System.out.println("See this character " + testUnicode1 + " and this character " +
```

```
testUnicode2); }
```

The output to the *console* is as follows.



- Boolean type

Unlike C / C ++, *boolean* type in Java language is only represented by two values: *true* and *false*. Therefore this data type is only used for checking logical conditions, not in computation, and you cannot assign an integer type to *boolean* as in C / C ++.

Declare a boolean as follows.

It's boolean male = true; correct boolean graduated = false;

Declaring a boolean variable as follows is false.

Wrong boolean male = 1;

If you do not declare a value for the variable, then the variable of type *boolean* will default to *false*.

II. Constant

Constants in English documents are called *const*, short for *constant*. A constant is similar to a variable, but especially if a variable is declared constant then it will *not be changed during the program*.

Because the above examples only help you declare a variable without changing the variable value, but in fact, a variable can be changed in value many times during the implementation of the variable. Application, you will get acquainted with changing the value of a variable in the following lessons. If a constant remains unchanged, if you intentionally change or reassign a new value after it is declared, you will receive an error message.

To declare a constant, you also declare it like a variable, but add *final* before the declaration. You will learn more about constants in a separate lesson on *final*. Now you can see the following example to get a quick understanding of constants.

```
final float PI = 3.14f;  
final char FIRST_CHARACTER = 'a'; final int VIP_TYPE = 1;
```

You should practice declaring constants with capital letters as in the above example, it helps us easily distinguish what is a variable and what is a constant later.

We just went over how to declare variables and constants. If you are still confused about the Java language in this lesson, it's just beginning, and it takes a lot of effort on your part. You will learn more about this Java language in the next lessons.

Lesson 5: Operator

If in *the previous post*, you are familiar with *declaring and using variables and constants* in Java. So today, we will learn how to apply these variables and constants to the computational logic in the program, through the study of *Operators*.

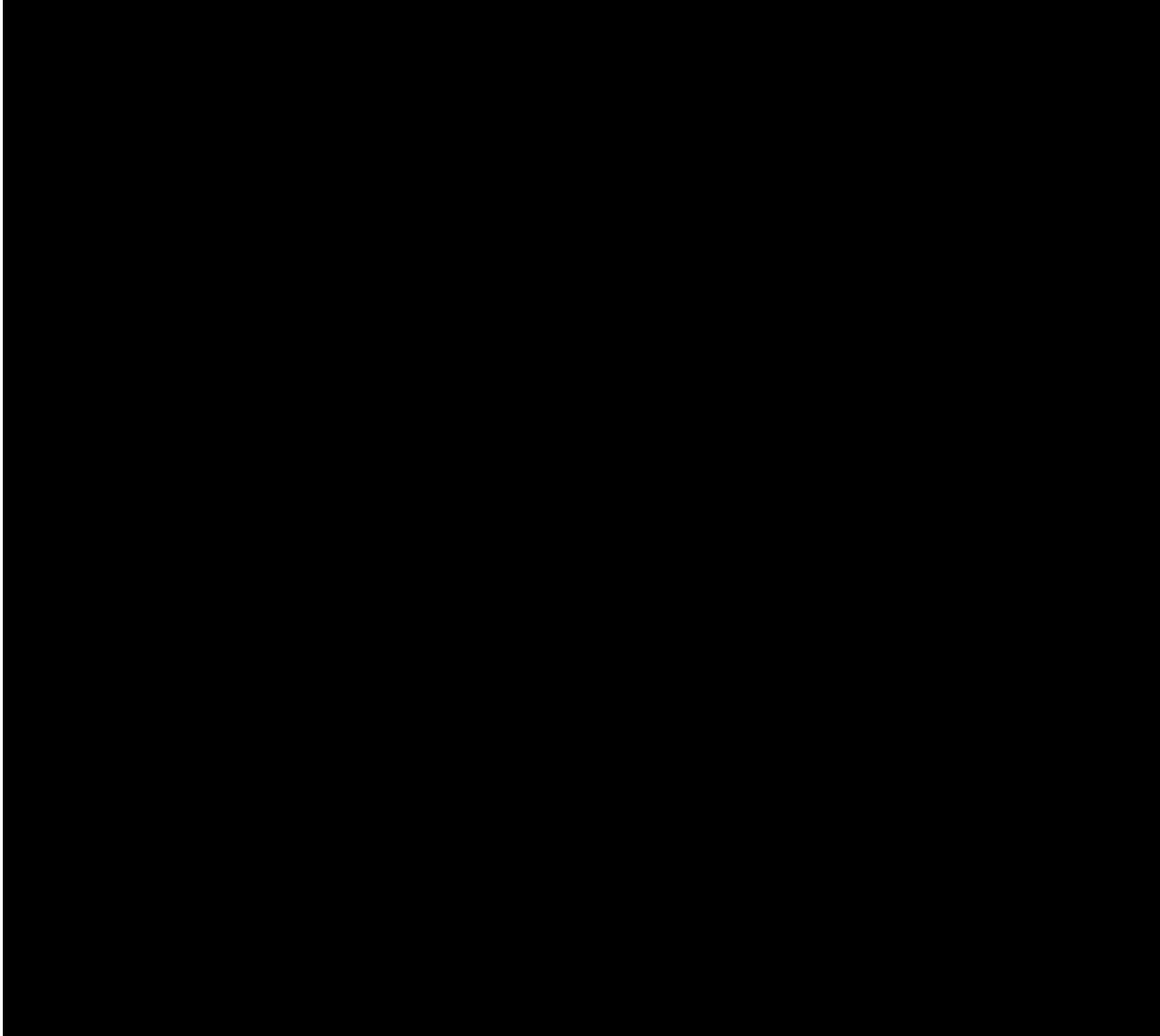
Before getting familiar with Operators, I want to talk about the concept of *Expressions* first.

I. Expression

You should know that programming is no different from doing math, all of us, Programmers, are simply applying the math that we have learned to do in programming applications. stop it. And to get acquainted with math knowledge,

let's return to the concept of *Expressions*. Mathematical definition of *expression* is “*a combination of operators and operand (Operand) accordance with a certain order* . In which *each operand can be a constant, a variable or another expression* ”.

Let me illustrate an Expression as follows.



Thereby * and + are Operators. 2 , y and 5 are operands, where 2 and 5 are Constants, and y is Variables.

As in math, you can use parentheses () to group the Expressions, and then the Expressions in these parentheses take precedence.

An illustrative example of an Expression with parentheses: $2 * (y + 5)$

II. Assignment Operator

Now let's move a little further from math to get into programming knowledge. In addition to the addition, subtraction, multiplication and division operators in mathematics, we also have many other typical operators of programming, one of the first we get acquainted with today is *Assignment Operator* .

In Java, the Assignment Operator is performed through the symbol "`=`". Very familiar, because in the previous post you have practiced declaring a variable as follows `int count = 5;` then the "`=`" sign here is an assignment. Assignment is defined through syntax.

`variable name = expression`

Meanwhile the results of a *EXPRESSION* (*EXPRESSION* could be a multi Math Expressions operand, or may just be a figure like the example in the previous post) will be assigned to *VARIABLE* . Ie *VARIABLE* will contain a value equal to the *EXPRESSION* bring through this assignment.

- Practice # 1

For this exercise, do the following assignment and show what the output log after assigning the value to the *thisYear* variable.

```
1 | final int THIS_YEAR = 2016;
2 | int thisYear = 2000;
3 | thisYear = THIS_YEAR;
4 |
5 | System.out.println("This year is " + thisYear);
```

The results are as follows, you should try running with Eclipse, or guess the result before clicking below.

1 | This year is 2016

- Assign Many Variables At The Same Time

In case you have the same value assigned to many different variables, instead of the example below you have to assign that value to each variable one by one.

```
1 | int x = 10;  
2 | int y = 10;  
3 | int z = 10;
```

Then you can perform the assignment with just one line as shown below.
Guaranteed correct.

```
int x;
```

It's correct int y;

```
int z;
```

```
x = y = z = 10;
```

Or you can write more concisely, this writing combines variable declarations with the same data type into the same line.

int x, y, z; It's correct

```
x = y = z = 10;
```

But note you cannot write like that. ^{Wrong} int x = int y = int z = 10;

III. Arithmetic Operator

We are familiar with the second Operator of programming, this operator is completely similar to mathematics, that is, *addition* , *subtraction* , *multiplication* , and *division* . These operations are grouped together into one Operator, called *Arithmetic Operator* . Specifically about the Arithmetic Operators I listed in the following table.

Operator Meaning +

Add Operator, can add numbers, and even add String will be discussed in detail *in the following series lesson* .

- Subtraction Operator.

* Multiplication Operator.

/

Division Operator takes the whole part, for example 5/2 will be 2, the remainder is removed.

%

Division Operator takes the remainder, for example `5% 2` will equal 1, which is the remainder of the division.

- Practice # 2

Try the code for the printout of the assignment from an Expression to the `age` variable with the Arithmetic Operators as follows.

```
1 | int age;
2 | int thisYear = 2016;
3 | int yearOfBirth = 1990;
4 | age = thisYear - yearOfBirth;
5 | System.out.println("I am " + age + " years old");
```

You can see the result by clicking on the item below.

1 | I am 26 years old

- Exercise Number 3

Try to guess the print result of the `num1` , `num2` , and `so3` as in the example below, and see if the result is correct as you guessed it.

`int num1; int num2; float num3;`

`num1 = 15 / 6; num2 = 15 % 6; so3 = 15 / 6;`

`System.out.println("The result num1 is " + num1 + ", num2 is " + num2 + ", num3 is " + num3);`

The result is num1 is 2, num2 is 3, num3 is 2.0

IV. Unary Operator

Another typical Operator of programming is the *Unary Operator* . With this Operator, only one Operator combined with an operand can produce the result. Please look at the following table.

Operator Meaning

Add Operator, this addition operator is different from the addition operator

+

in the *Arithmetic* table above, which represents positive numbers. Since normal positive numbers don't need to display this operator, you won't see its use either.

-

Subtraction Operator, in contrast to the addition operator above, this subtraction operator represents negative numbers.

- ++ Increment Operator, which increases the value of the operand by 1 unit.
- Decrement Operator, which reduces the value of the operand by 1 unit.

Logical Negation Operators. This operator will reverse the value of the logical ! expression variable. If the logical expression is *true* , it will be reversed to *false* and vice versa

- Exercise Number 4

Look at the expression below, *num* will be assigned an Expression, where *positiveNum* is added a *Unary Operator* representing negative numbers.

```
first int positiveNum = 4;
```

```
2 int num = positiveNum + -10;
```

3 System.out.println("The result is " + num); And the result as follows is easy to guess. (-6)

- Exercise Number 5

Pay attention to the following "++" and "--" Operators, as mentioned, they will increase or decrease the operand after it by 1 before performing other operations.
first

```
int num1 = 4;
```

```
int num2 = 10;2 int reuslt = ++num1 + --num1;3 System.out.println("num1 is " + num1 + ",  
num2 is " + num2 4 + ", result is " + reuslt);
```

And the result is,

num1 is 5, num2 is 9, result is 14

I would like to explain a little bit in Exercise # 5 above, because the Unary Operator of this form will be a bit strange to newcomers to programming. With the implementation of Expression *result* = *++ num1 + - -num2*; The system will execute the expression *++ num1* first and *num1* then carry the value 5 (*++ num1* at that time is similar to the expression *num1 = num1 + 1*; so), then the system also executes *-- num2* gives the value 9 (similarly *-- num2* will be the same as *num2= num2- 1*;), finally add these two numbers and assign it to the variable *result* .

Have you ever wondered that the two ways of writing *++ num1* and *num1 ++*

are different? The answer is yes. Just like C / C ++, when you write `++ num1` , Java will perform an increment of `num1` before using that value into Expression. If written `num1 ++` , Java will take the value of `num1` used in the Expression before increasing its value.

To better understand, try to fix the above code a bit as follows and run it again. first

```
int num1 = 4;  
int num2 = 10;2 int soKetQua = num1++ + num2;3 System.out.println("num1 is " + num1 +  
", num2 la " + num2 4 + ",result la " + result);
```

The output is.

num1 is 5, num2 is 10, result is 14

With the above test, you can see that `num1 ++` is not applied at all, increasing the value of `num1` to the expression, but `num1` still carries the value 4 and then plus `num2` is 10 , resulting in 14 . After executing the Expression, `num1` will be incremented and so you can see that `num1` is 5 .

V. Relational Operator

The Relational Operator compares the Expressions against each other to yield a *boolean* value, that is, either *true* or *false* . Look at the table for the following Relational Operators.

- Exercise 6

Try to guess if the output will be *true* or *false*

`double weight = 71.23;`

Operator Meaning

`==` Equal

`!=` Not equal (other)

`>` Bigger

`>=` Greater than or equal

`<` Less

`<=` Less than or equal

```
int height = 191;
```

```
boolean married = false; boolean attached = false; char gender = 'm';
```

```
System.out.println(Result 1: " + (!married == attached));
```

```
System.out.println("Result 2: " + (gender != 'f'));
```

```
System.out.println("Result 3: " + (height >= 180));
```

```
System.out.println("Result 4: " + (weight > 90));
```

Do you have your own results yet? If so, click on the results below to compare.

Result 1: false

Result 2: true

Result 3: true

Result 4: false

VI. Conditional Operators

Operator Meaning

&& Compare AND.

|| Compare OR.

?:

Conditional comparison, will be discussed when we learn how to write *if..else..* in *the next lesson* .

The *Conditional Operators* compare Expressions that are *true* and *false* . This operator is also known as *the Logical Operator* . You just imagine this operator would behave like the saying "*if it doesn't rain and I have money I'll go out today*" , where "*if it doesn't rain*" carries a *boolean* value, "*I have money*" Is also a *boolean* value, which is compared by the " *and* " operator. So if "*it doesn't rain*" and "*I have money*" both are *true* then the comparison result will be *true* , ie "*I'm going out today*" is *true* , meaning I will go out. Conversely, if one of the first two sides is *false* then the result will be *false* , I will not go out. The Condition Operators are shown in the above table.

The following is the *Table of Values* , i.e. the results comparing the conditions of the Operators && and || as follows.

p q p && q p || q

false false false false

false true false true

true false false true true true true

• Practice # 7

Try typing the following commands and contemplate its results.

```
int age = 18;
```

```
double weight = 71.23; int height = 191;
```

```

boolean married = false; boolean attached = false; char gender = 'm';

System.out.println(!married && !attached && (gender == 'm'));
System.out.println(married && (gender == 'f'));
System.out.println((height >= 180) && (weight >= 65) &&

(weight <= 80));
System.out.println((height >= 180) || (weight >= 90));
The results is:
true
false
true
false

```

We try to explain the first line in log to understand better. First turn *married* value *false* , *! Married* to reverse the value to *true* . Similarly, *attached* is *false* and *! Attached* is *true* , *(gender == 'm')* will be *true* . In summary we have *true && true && true* , and the final result in the truth table is *true* .
On the second line, prioritize the expression in parentheses first *(gender == 'f')* will return *false* . The variable *married* is declared *false* . *false && false* results in the truth table as *false* .

You go on for the same explanation for the rest of the lines.

VII. Bitwise Operator

The Bitwise Operator is the Operator that will interact directly with the bits of the operand. To understand what working with bits is very lengthy, maybe I will talk more clearly in other articles. But you can understand that the essence of all data that computers can understand and work with is binary data represented by only two values of *0* and *1* . Each of such a value of *0* or *1* is a *bit* , so it can be said that the Bitwise Operator is the operator that directly interacts with the *bits* .

Note that this Bitwise Operator will not work with data of real number type (*float* and *double*).

The following table is the Bitwise Operator.

Operator Meaning

& AND

| OR

\wedge XOR

\sim NOT, reverses bit 0 to 1, and vice versa 1 to 0.

>> Right translation

<< Left shift

Remember, AND in the Bitwise Operator has only one $\&$, and AND in the Conditional Operator has two ampersands and $\&\&$. And one more difference is that $\&\&$ only compares *boolean Expressions* to each other, $\&$ compares based on bits, and so $\&$ can work with all primitive data types in Java except for the real number said on. Similar for differences between $\|$ and $|$.

So before getting into the practice of Bitwise Operator, you should take a look at *the Truth Table* for Bitwise as follows.

p q p $\&$ q p | q p \wedge q

0 0 0 0 0

0 first 0 first first

first 0 0 first first

first first first first 0

• Exercise Number 8

In this exercise, we share code to try the Bitwise Operators as follows, I will explain why the result below this exercise.

1 byte num1 = 5;

2 byte num2 = 12;

3

4 System.out.println("Operator &: " + (num1 $\&$ num2));

5 System.out.println("Operator |: " + (num1 | num2));

6 System.out.println("Operator ^: " + (num1 \wedge num2));

7 System.out.println("Operator ~: " + (num1 $\&$ \sim num2));

8 System.out.println("Operator <<: " + (num1 << 1));

9 System.out.println("Operator >>: " + (num2 >> 2)); The results are below.

1 Operator &: 4

2 Operator |: 13

3 Operator ^: 9

4 Operator ~: 1

5 Operator <<: 10

6 Operator >>: 3

Now let's explain what the math means. We all know that the *byte* datatype is allocated by the system for 1 *byte* of memory, and 1 *byte* = 8 *bits* , so the variables *num1* and *num2* with values 5 and 12 are represented as bits, respectively.

$$5 = 0000\ 0101\ 12 = 0000\ 1100$$

num1 & num2 : if you take *0000 0101 & 0000 1100* , you apply the truth table to compare each bit with each other, you will get the result *0000 0100* , this is a bit representation of the number 4 , and is the result of the first log line.

num1 | num2 : same as above *0000 0101 | 0000 1100* would be *0000 1101* , which is the bit form of the number 13 .

num1 ^ num2 : *0000 0101 ^ 0000 1100* would be *0000 1001* , which is the bit form of the number 9 .

num1 & ~ num2 : we consider \sim *num2* , \sim is the *NOT operator* , it will reverse the bits of *num2* to *1111 0011* , so *num1 & ~ num2* will be *0000 0101 & 1111 0011* , would be *0000 0001* , which is a bit form of number 1 .

num1 << 1 : means taking the bit sequence of *num1* to move 1 bit left, it becomes *0000 1010* , which is a bit form of the number 10 .

num2 >> 2 : that is, taking the bit sequence of *num2* must move 2 bits, it becomes *0000 0011* , which is a bit form of the number 3 .

We have just gone through the lesson of Operators in Java, through this lesson, you have a better understanding of the Java language, right? This is Operators will be pretty much applied to the knowledge and practice in the future. Please try to follow the interesting lessons ahead.

Lesson 6: Type Casting & Comment Source Code

This lesson we will talk about 2 "small" issues , that is *casting* and *commenting source code* , you should also know that small here is small about the total number of writing, but in fact, the two issues in this lesson are very important. important for the next lessons.

First of all, once again, we have been familiar with *declaring a variable (or constant)* , then you *need to specify a data type for that variable or constant*

before using it. Such an initial data type is static. This means that if you define the variable as an *int*, it will always be of type *int*, if you define it as a *float*, it will always be of type *float*. Have you ever wondered if you take these two variables with different data types together, will it create a variable of what type? And can we change the data type of a variable, or a declared value?

In this lesson we will answer the above questions through the concept of *Type casting*.

I. Casting Concept

As I mentioned above, you probably have a rough idea of it. Specifically, casting is a form of *converting the data type of a variable into a new variable with another data type*. So this casting doesn't change the data type of the old variable, it just helps you create a new variable with the new data type. The concept is so, and what is the purpose, please continue reading this lesson.

Remember that since you are only new to *primitive data types*, casting only refers to primitive data type casting.

1. Types Of Casting

If we classify casting based on variable storage capacity, we have the following two types of casting.

- *Expanding storage capacity* : this compression will convert data from data type of smaller size to data type with larger size. This does *not lose the value of the data*. In the initial example, you have a variable of type *int*, the value is 6, you force data from type *int* to *float*, and then assign it to the new variable *float*, then the new variable will *float* with the value 6.0. Forcing this type of casting is normally left to the system by *default*. You will understand what the implications are below

- *Narrow storage capacity* : the cast will make data conversion from data type larger sizing data types into a smaller size. This *can cause loss of data value*. For example, you initially have a variable of type *float*, the value is 6.5, you squeeze data from type *float* to *int*, then assign it to the new variable *int*, then the new variable *int* will hold the value 6. This casting cannot be done by the system by default, at this point the system will error, you must perform *explicit casting* for it.

So, for ease of remembering, you can classify casting into two types, which are

implicit and *explicit* .

- *Implicit casting*: this casting can take place automatically by the system, when the system knows when to extend storage capacity, it will. As I mentioned above, it is this type of expansion that will *not lose the value of the variable* . When starting to perform casting, the system will check the following rules, if it is satisfied, it will be pressed.

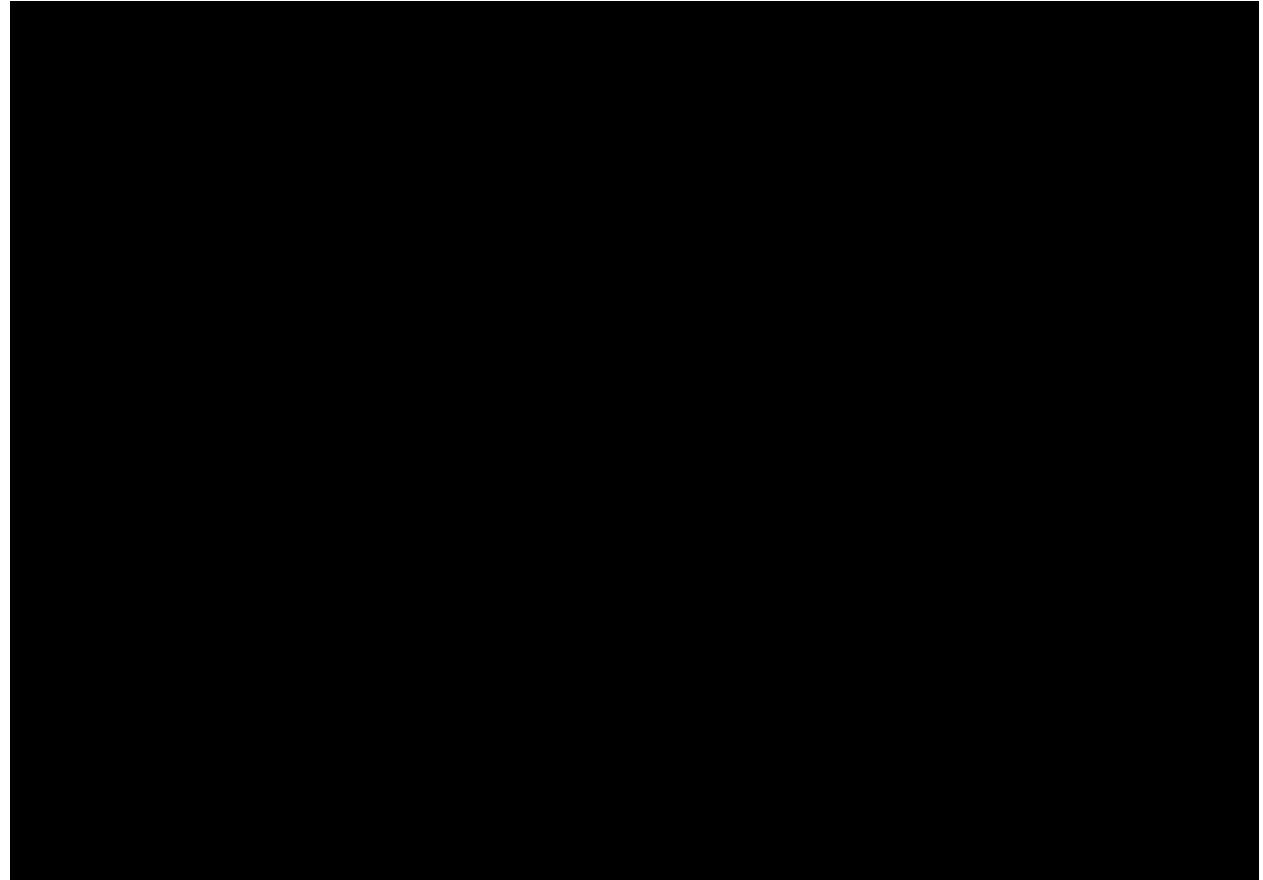
+ *byte* can be cast to *short , int , long , float , double* . + *short* can be cast to *int , long , float , double* . + *int* can be cast to *long , float , and double* . + *long* can be cast to *float , double* .
+ *float* can be cast to *double* .

- *Explicit casting* : when the conditions are not met to automatically cast, the system will report an error. The rest is that you must explicitly specify casting. Since this type of casting can *lose the value of the data* , it is very important for you to make a decision, but the system does not dare to decide. You would specify the following when you want to perform this explicit casting.

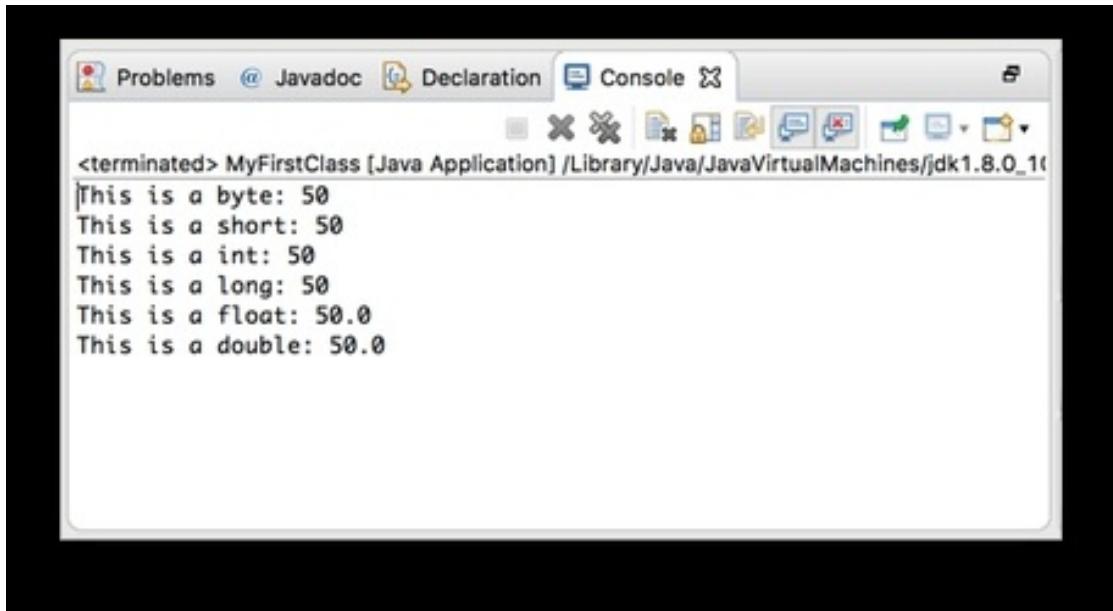
(duplicate_data type) variable_name;

2. Example Implicit Casting

With this type of casting, you notice that we do not need to do anything, just keep coding and the system will perform the casting by itself. Take a look at the following code.



You see, with *i* initially declared as *byte* type with the value 50 , after the assignment to variable *j* (the value is *short*), the value 50 in this expression is automatically converted to the higher *short* data type and assigned. into variable *j* . The same goes for assignments to *k* , *l* , *m* , *n* . The output is as follows.



```
<terminated> MyFirstClass [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101
This is a byte: 50
This is a short: 50
This is a int: 50
This is a long: 50
This is a float: 50.0
This is a double: 50.0
```

But what if you test by asking the system to force a larger data type into a smaller one. The system will immediately report an error. And so you need to intervene next item below.

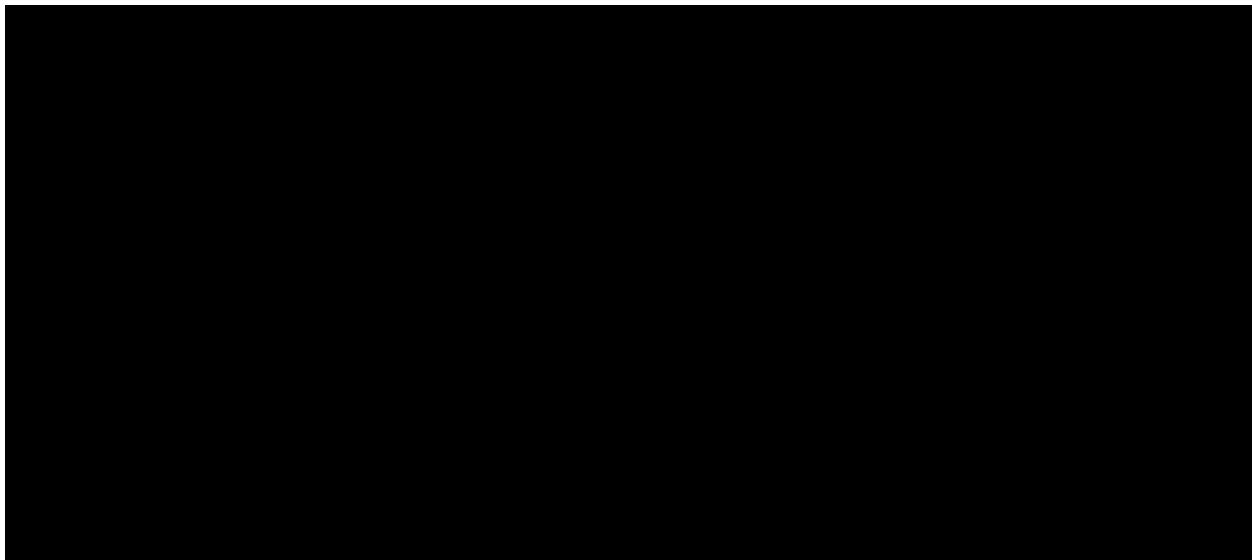
```
int i = 50;
short j = i;
```

3. Example Explicit Pressing

Returning to the above error example, the system refused to automatically cast the default type, so if you still want to explicitly cast it, now you need to use the cast structure. I have given it above, like this.

Take a look at the line `short j = (short) i;` This line will force the value of `i` to `short` and assign it to variable `j`. The only thing you designate with `(short)` looks is the idea immediately, so the name is explicit.

Take a look at a more realistic example, in some cases where you want to remove the decimal point of a real number type, the fastest way to do this is to cast this real number to an integer type. This type of casting is actually ... deliberately devaluing the variable.



The result of this casting is printed as follows, please pay attention to the line that prints out type `int`, losing the decimal part.

A screenshot of a Java IDE interface, specifically showing the 'Console' tab. The title bar includes tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. Below the tabs is a toolbar with various icons. The main console window displays the following text:
<terminated> MyFirstClass [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_10
This is a int: 7
This is a double: 7.5

II. Comment Source Code

Because my knowledge of casting is too short, I always talk about *Comments* in this post. The two knowledge about casting and commenting it doesn't mix anything, but it will help you in the code process.

The reason I use the word comment instead of translating it into Vietnamese is to avoid misunderstanding, because most of you know that comment means "*comment*" , but actually the purpose of comment in coding is "*write Note*" or "*comment*" , it helps you explain some lines of code, you can comment comfortably anywhere in the source code, the compiler will ignore these comments when building, so there will be no comments. errors happen to them.

1. Comment Method

You have 3 ways to comment as follows.

- // *text* - When the compiler encounters the // symbol , it ignores all characters from // to the last character of that line. So with each // will help you comment 1 line.
- /* *text* */ - When the compiler encounters the / * symbol , it will ignore all characters from / * through * /. So this way of commenting can help you comment on multiple lines at the same time.
- /** *document* */ - Same as above, but notice that there are two * When started, this comment helps to extract the documentation. It is called document. This comment will be more clearly stated in another post.

2. Example Comment

You look at the example "3 in 1" below, I have included all three ways of comment above mentioned above into a program. In fact, you do not need to comment too much as for example, you should comment when it is necessary to explain to a certain strange line of code, or note the author of that line of code, or the date of the modification of this code, ...

```
public class MyFirstClass {  
  
    /**  
     * Document comment will be introduced latter *  
     * @param args
```

```
*/  
  
public static void main(String[] args) {  
  
    // One line comment  
    // Comment this line  
    double d = 7.5;  
    int i = (int) d; // Cast the double into int  
  
    /*  
     *The bellow code will be in console, * and you can keep writing comment *  
     *using enter key  
    */  
  
    System.out.println("This is a int: " + i); System.out.println("This is a double: " +  
    d); }  
}
```

You have just gone through Java casting methods, and how to comment source code. This lesson is easy but very important, you will use casting and commenting many times in your projects.

Lesson 7: Import / Export on Console

In fact, in this lesson I will talk a bit further than the knowledge you have been familiar with in the previous lessons, this knowledge has much to do with OOP (object oriented) that you will learn in the following lessons.

Oh don't be discouraged yet. You can see how this lesson will guide you to use a "tool" for you to import / export through the *Console* . This tool helps you after entering data from the keyboard, it will take this imported data and transfer it to your program, for the program to process, and then output the results via the *Console* screen .

After in this lesson you can comfortably test your lines of code, to better support your Java learning process. But first, let's take a deeper look at this Console concept.

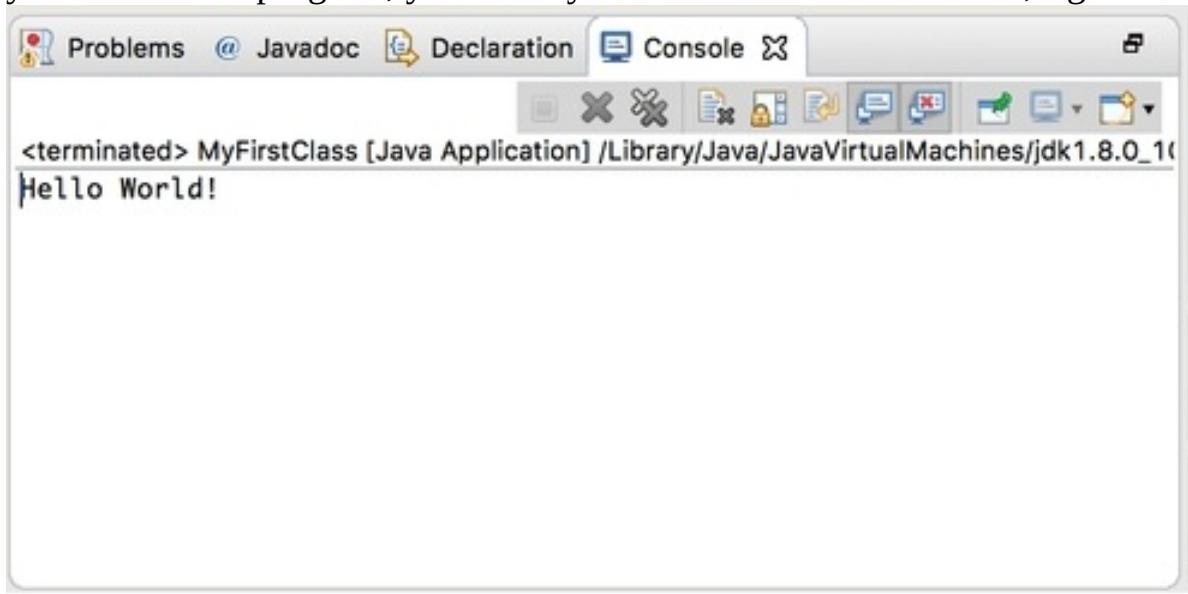
I. Console Concept

First, you can understand the *Console* as a dedicated console, as the Game Console is the specialized device for game control, or management websites like employee management can also be considered as Console. dedicated to management.

In the programming language, Console is known as a pure way of controlling an application through text (command line), it is distinguished from control by UI.

By the way, I want to make it clear, that all the lessons in *our Java learning program* will be based on *console* control , so it will be different from Java applications with specific interfaces. , the lessons mainly give you knowledge of how to use Java language (to be able to program Android applications), rather than programming Java applications, so there will be no spectacular interface. You should remember this.

And the *Console* in our learning program is here, in the previous tutorials, when you execute the program, you already know this *Console* window , right.



II. Enter On Consolse

We talked about the ways in which you import data from the *Console* first. To get started, open Eclipse up. First, I want you to try declaring a *Scanner object* like the code below. You code away then I will say more clearly why you must code like this below this line of code.

```
Scanner scanner = new Scanner(System.in);
```

The concept of *class* (*class*) or *object* do something, you will get acquainted later, it belongs to his knowledge of OOP as having said to head the unit. But as you can understand in advance that *an object* is like a *variable* , you can name it arbitrarily (according to the *variable's naming rule*). As above we name the *object* of Scanner *scanner* . The only difference is that the *object of the class* must be initialized with the keyword *new* as above. Then your overall code is as follows.

```
public class MyFirstClass {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Scanner scanner = new Scanner(System.in);  
  
    }  
}
```

1. *Code Faster*

This section allows me to ramble on a little coding experience. That is, after many of you have finished typing the above code, there may be an error from the system as shown below.



The error is because the system is not knowing what the *Scanner* class is. That's because in the past you only used *primitive* variables , or classes in the *java.lang* package , that we don't care about. Well then what should we care? In fact, in Java, every time you code, you have to specify the origin of the classes you're going to use, through the *import* keyword at the top of the file, followed by *import the* package containing the class to be used. You can see more knowledge about packages [here](#) .

Accordingly, in the above example we need the *Scanner* class . Therefore, we need to import the correct package that the system wants.

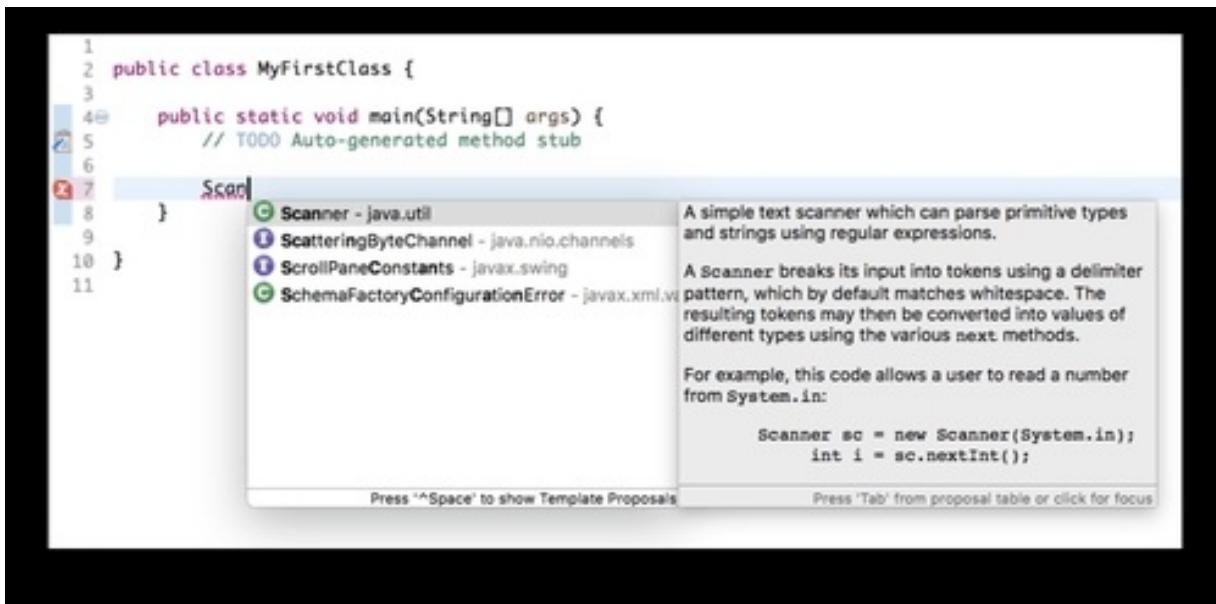
To import the correct package with the *Scanner* class you need to use above is extremely easy, there are many ways to do this, please choose one of the following ways.

a. Crl + Space

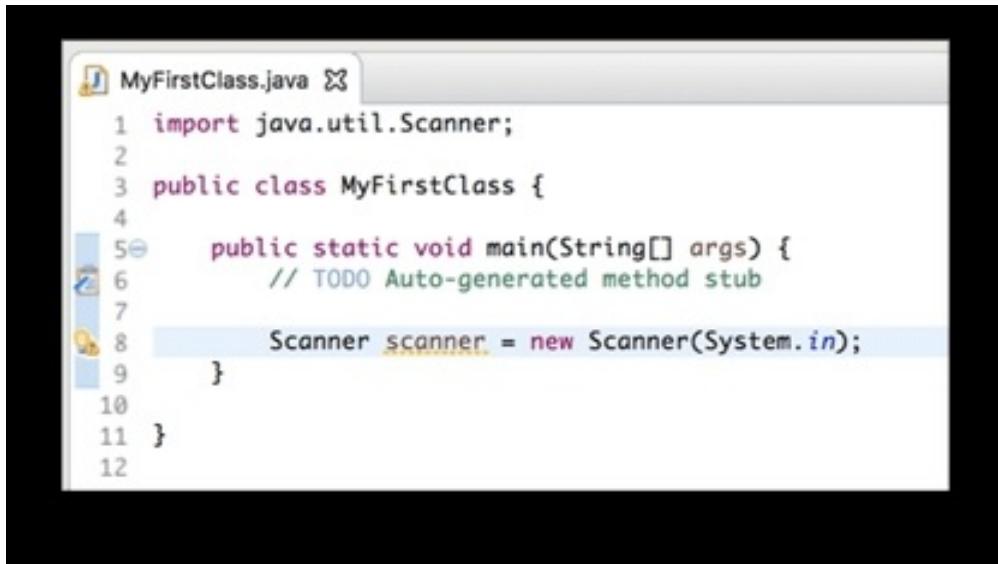
This divine key combination programmer must also know, when you are typing

a certain class of the system that wants them to be automatically filled in the remaining letters, and of course with Eclipse this key combination will also promote. allows you to import the package of that object automatically.

Suppose you are typing the *Scanner* declaration code above, instead of typing everything like the sample code above, try typing a few words and then press *Ctr + Space*, you will see suggestions from the system, like when you press asterisk (*.*) in *the first lines of code* you are familiar with.



So with the light trail in the first line *Scanner*, you press *Enter*, the result of the code is automatically completed and the system also imports the *java.util.Scanner* package for you without having to remember who they are and from where. Come on, right. You try to type all the above commands, but this time diligently press *Ctr + Space* in each word (*scanner*, *new*, *Scanner*, *System*). The results will be error free.

A screenshot of the Eclipse Java IDE interface. The title bar says "MyFirstClass.java". The code editor contains the following Java code:

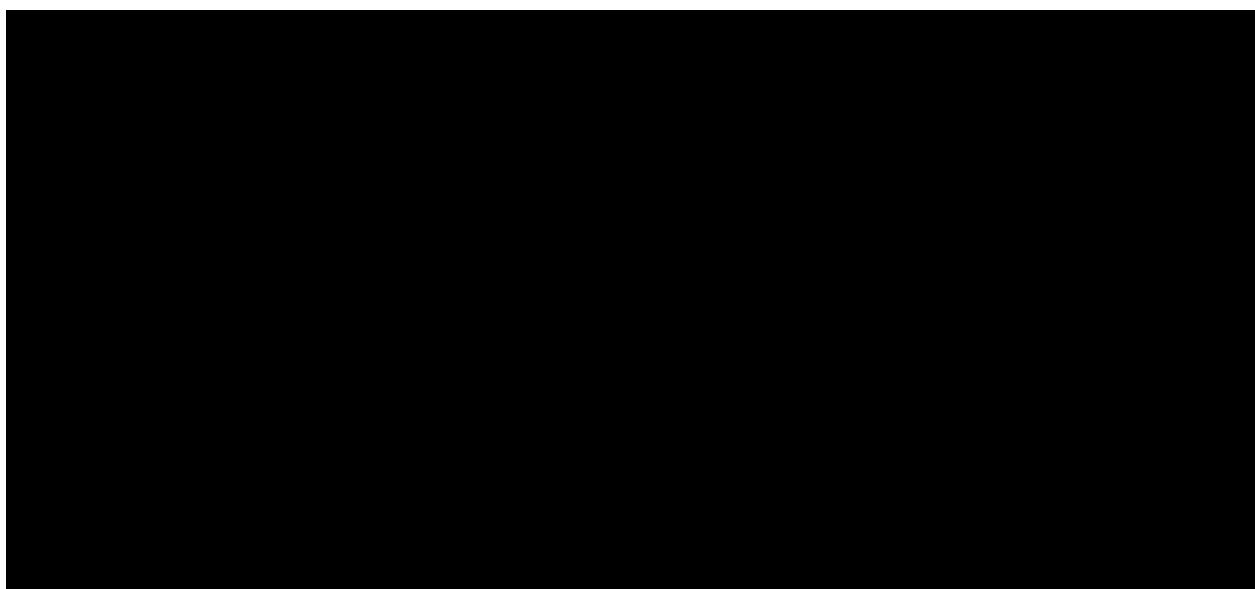
```
1 import java.util.Scanner;
2
3 public class MyFirstClass {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7
8         Scanner scanner = new Scanner(System.in);
9     }
10
11 }
12
```

The word "Scanner" in the line "Scanner scanner = new Scanner(System.in);" is highlighted in yellow, indicating it is a suggestion or a potential error. A light blue tooltip-like box appears to the right of the word, containing the text "Scanner" and several other options like "Scanner<T>" and "Scanner<T>".

b. Using Hints

From Eclipse

This second way is to keep typing, and then every time you type something wrong, don't worry. You should pay attention if the left bar of the editor appears a light bulb with a red X , when Eclipse has identified the error and knows how to fix it, you just need to click on this light bulb. See the following error correction suggestions.



What are you waiting for without choosing *Import 'Scanner' (java.util)* , this means that Eclipse asked if you allowed to import the *java.util.Scanner* package or not.

c. Manually Import

Of course, this way for you to know exactly what package this class is in, so you can type import manually.

Stroll a bit long, back to the fact that you just declared the object of the *Scanner* class above. On the next line you just need to call *scanner.nextInt()*; then when you run the program, when the system executes on this line, it will stop and wait, and then the *Console* will appear a blink waiting for the user to enter a value of data type *Xxx* and then proceed to assign. This value goes to the corresponding variable and continues the commands below.

Try the following examples for each *scanner.nextInt()*; Be specific.

2. String Type Data Import Example

For this example you try for users to enter their name from the *Console* and then print the hello right on the *Console* as follows.

To wait for the user to enter the name you type the following after the *scanner* declaration .

```
String name = scanner.nextLine();
```

Then the username entered from the *Console* will be assigned to a variable of type *String* (this is the *type of string that you will be familiar with*) named *name* . But to make it easier for the user, we should have the *System.out* lines to print out instructions for the user, and print the greeting line at the end of the program. Our overall code is as follows.

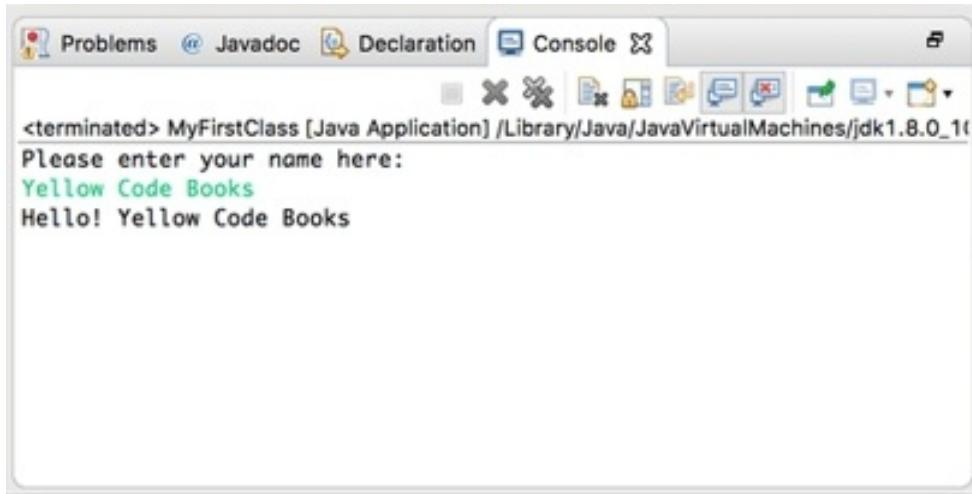
```
import java.util.Scanner;;
public class MyFirstClass {
    public static void main(String[] args) { // TODO Auto-generated method stub

        Scanner scanner = new Scanner(System.in); System.out.println("Please enter
        your name here: "); String name = scanner.nextLine();
        System.out.println("Hello! " + name);

    }
}
```

At this time, if you run the program, only the text "*Please enter your name here:*" appears, don't turn off the *Console* window , continue typing in a text, after *Enter* you will receive another text. with the text "*Hello!*" and the text you just entered, the text that prints out is the content of the variable name that the

scanner takes data from the *Console* and assigns to it.



3. Data Input

Example Type int

Similarly if this example asks the user to enter the age, you should also have a statement that prints out a hint, the line *scanner.nextInt()* , and the line that prints the output as follows.

```
System.out.println("How old are you? "); int age = scanner.nextInt();
System.out.println("Your age is: " + age);
```

4. Float Data Import Example

Scanner supports import for all primitive data types, this example imports *float* type and you can apply to the remaining primitive data types.

```
System.out.println("How about your salary? ");
float salary = scanner.nextFloat();
System.out.println("Your salary is: " + salary);
```

III. Exporting on Console

As you probably know, to output the data to the *Console* , just call *System.out.println()* . So easy, the previous lessons and in this lesson you are familiar. But the knowledge about exporting data to *Console* still has some fun things, I would like to list them briefly as follows.

You can use *print()* instead of *println()* . If with *println()* , you already know it helps to output data and then attach to the row after export. Then *print()* only

outputs data and does not drop. You can try applying `print()` to instructional messages like “*Please enter your name here:*” *up there* to see the difference.

If you want to display the following special characters: single quote (‘) , double quote (“) , and slash (\) . Then you just enclose a slash (\) in front of these characters. For example, you want to display the line *The directory containing the file "that" is C:\Location\Ay* , then type the following command `System.out.print ("The directory containing the file \" that \" is C:\\Location\\Ay");` .

In addition, you can also insert characters to help format the output, such as “\t” will insert a tab, or “\n” will help a newline. You try to verify yourself by typing this command line `System.out.print ("\tHello\nWorld");` .

Since we have been practicing the worthwhile I / O commands from the console, let's see how they help you in the next lessons.

Lesson 8: Conditional Statements

And this's lesson we will talk to the *Statements Flow Control (Flow Control)* . I would also like to say first that since these commands are a lot and important, I split them into two groups. The first group contains the *Conditional Statements* (also called *Branch Statements*) that you will be familiar with this. The other group is *the Repeat Statement* you will be familiar with in the following this lesson.

Let us first talk about the general concept of the two groups, what is the concept of a *Flow Control Command* .

I. Flow Control Command Concept

To make this concept easier to understand, remember to see your code in the previous lessons (although we haven't coded much yet), you can see when you code and the lines of code are covered by Eclipse. executed, they will be read and executed by this compiler linearly from top to bottom, right, from line 1 to the last line.

In fact, when you need a more complex algorithm for your application, such as needing to access the database and print to the console every single student

information, then the implementation of linear code line by line will be extremely complicated, you have to write thousands of lines of code for sequential reading of thousands of students in the database. And yet, if for each student to be read out there are certain conditions, such as only printing out the number of female students, then coding and executing linearly is a nightmare.

That is why the *Flow Control Statements* are created by languages, to create a new thread of execution, which can be an iterative thread, or a branch thread, so that they can direct the real compiler. examining a certain piece of code many times, or omitting not executing certain code, ... As mentioned, in this's lesson you are familiar with the first group in the *Flow Control Statement*, which is the group of *Conditional Statements*. Help with channel branching.

Before getting familiar with the commands, I would like to start to talk about two "divine" symbols that from the first lesson you have met, these two symbols are very helpful for this's lesson and coding. Your friends later, it is the symbol { and }, the curly braces help to form a *Block of Commands* (also known as *Block*).

II. Block Concept

As you just know, the *Block* in Java is denoted by curly braces ({ and }).

Going back to the past, you will see that this parenthesis appears in the class declaration (you will learn about the class in the following OOP lessons), in this case the brackets created a block. It acts as a wrapper for the code and indicates that all of the code inside it is class code, then they (those codes) must follow the rules of the class (you will know these rules later). Any lines of code outside the curly braces will not be under the control of that class. The curly braces that I mentioned appear as shown below.

A screenshot of a Java code editor window titled "MyFirstClass.java". The code defines a class named "MyFirstClass" with a main method. Two curly braces are circled in red: the opening brace at line 3 and the closing brace at line 14. The code is as follows:

```
1 import java.util.Scanner;;
2
3 public class MyFirstClass {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7
8         Scanner scanner = new Scanner(System.in);
9         System.out.println("Please enter your name here: ");
10        String name = scanner.nextLine();
11        System.out.println("Hello! " + name);
12    }
13
14 }
```

Or the curly braces appear in the method declaration (you will also learn about the method in the next lesson), which helps to create a block of code that acts as a wrapper for the code that shows that all the code inside it is the code of the method. wake up that. As above, every line of code outside of the method's curly braces falls outside the method's processing logic. The method braces appear as follows.

A screenshot of a Java code editor window titled "MyFirstClass.java". The code defines a class named "MyFirstClass" with a main method. Two curly braces are circled in red: the opening brace at line 5 and the closing brace at line 14. The code is as follows:

```
1 import java.util.Scanner;;
2
3 public class MyFirstClass {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7
8         Scanner scanner = new Scanner(System.in);
9         System.out.println("Please enter your name here: ");
10        String name = scanner.nextLine();
11        System.out.println("Hello! " + name);
12    }
13
14 }
```

In addition to the parentheses of the class and the method, you can also create any block of code in your lines of code, just enclose those statements in a pair of curly braces. Creating blocks like this can help keep the lines of code organized.

And obviously the block is also applied to *Conditional Statements* that we will get familiar with below. That is why we need to be familiar with the block before going into the official lesson.

But no matter what the block is used, you must remember one thing, if there is a { declaration to start a block, there must be a } somewhere to close the block. . If a program has a total number of { not equal to the total number of marks }, an error will occur.

III. Scope of a variable

We get acquainted with one more knowledge. Because when you are familiar with the *Block* , you should also know the *Scope of the variable* , the scope of the variable will be greatly affected based on these blocks. Sounds familiar, but strange, right, because you certainly know the *variable* , your task this is to know about its scope as well. How do we define the scope of the variable? Actually I also have read many documents on this issue, there are many ways to

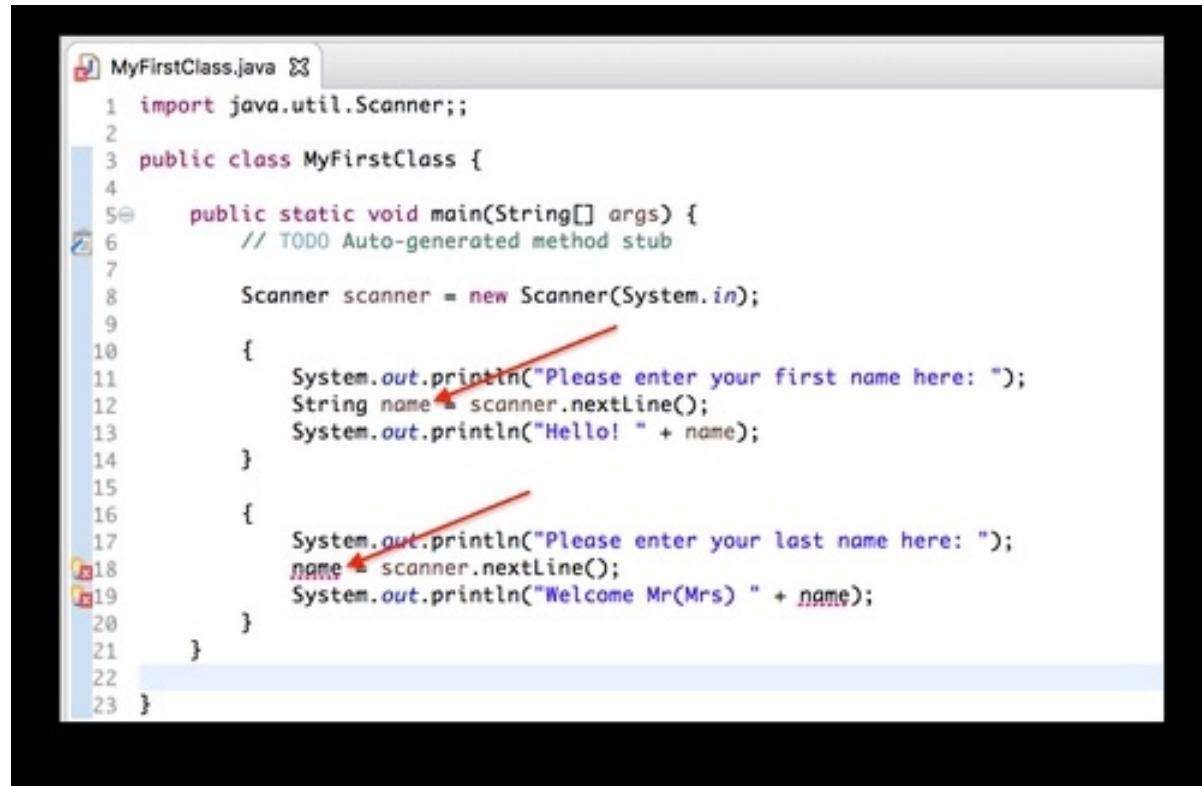
define the scope, but how to determine the most intuitive is probably differentiate the scope of variables based on the *impact local or global* c ũa it.

With local scope, which is the scope where the variable is *only active locally in a block*, it cannot be used outside of the block.

As for the global scope, which is the scope where the variable is declared in the external block, then *the blocks inside that block can also use this global variable*.

In the example below, you can see that the variable *name* is declared by you in a block, so it is a local variable of that block, you cannot reuse this variable in another block (you can see the system error as shown below). You can only use this *name* variable if you declare the variable in another block, but note that then two variables *name* in two different blocks of code will have nothing to do with each other.

F



The screenshot shows a Java code editor window with the file "MyFirstClass.java" open. The code is as follows:

```
1 import java.util.Scanner;
2
3 public class MyFirstClass {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7
8         Scanner scanner = new Scanner(System.in);
9
10        {
11            System.out.println("Please enter your first name here: ");
12            String name = scanner.nextLine();
13            System.out.println("Hello! " + name);
14        }
15
16        {
17            System.out.println("Please enter your last name here: ");
18            name = scanner.nextLine();
19            System.out.println("Welcome Mr(Mrs) " + name);
20        }
21    }
22
23 }
```

Two instances of the variable *name* are highlighted with red arrows pointing to them. The first instance is located at line 12, and the second is at line 18. Both instances refer to the same local variable declared in the outer block (line 10), which causes a compilation error because the variable is already defined.

Also in this example, but I declare the variable *name* outside the command block, then this *name* variable is considered a global variable of two sub-blocks, and so it can be used freely without error.

```
MyFirstClass.java
1 import java.util.Scanner;;
2
3 public class MyFirstClass {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7
8         Scanner scanner = new Scanner(System.in);
9         String name;
10
11     {
12         System.out.println("Please enter your first name here: ");
13         name = scanner.nextLine();
14         System.out.println("Hello! " + name);
15     }
16
17     {
18         System.out.println("Please enter your last name here: ");
19         name = scanner.nextLine();
20         System.out.println("Welcome Mr(Mrs) " + name);
21     }
22 }
23
24 }
```

Being encroached by the country, I continue with the example with the variable *name* placed outside the *main () method*, this time it will be considered a global variable of all the methods in this class (not just the *main* method. Where offline, and you also do not pay attention to the *static* declaration of the variable, this declaration will be discussed in *this lesson* when you learn to OOP).

```
MyFirstClass.java
1 import java.util.Scanner;;
2
3 public class MyFirstClass {
4
5     static String name;
6
7     public static void main(String[] args) {
8         // TODO Auto-generated method stub
9
10        Scanner scanner = new Scanner(System.in);
11
12        {
13            System.out.println("Please enter your first name here: ");
14            name = scanner.nextLine();
15            System.out.println("Hello! " + name);
16        }
17
18        {
19            System.out.println("Please enter your last name here: ");
20            name = scanner.nextLine();
21            System.out.println("Welcome Mr(Mrs) " + name);
22        }
23    }
24
25 }
```

IV. The if statement

With the mustache finished, we begin to get into the first Conditional Statement, *the if Statement*. Only with the name *if* but in fact there are four statements of this form that you must grasp, that is: *if* , *if else* , *if else if* , and *?:* . We begin to get acquainted with each statement as follows.

1. *if*

Syntax for the *if statement* .

```
if (conditional_expression) {
commands;
}
```

Which *conditional_expression* is an expression that returns the result as a value *boolean* . *COMMANDS* will be executed only when *conditional_expression* return the value *to true* only. You see that the block has been applied to wrap *commands* .

Example for *if statement* .

```
Scanner scanner = new Scanner(System.in); System.out.println("Please enter  
your age: "); int age = scanner.nextInt();
```

```
if (age < 18) {  
    System.out.printf("You can not access"); }
```

Alone explain a little example above, you see *conditional_expression* now very simple, if the variable *age* the user entered from the keyboard is smaller than *18* , then this expression returns *true* , then the statement block of commands *if* this Only one line prints out to the console an underage message, this line of code will be executed. If the user enters an *age* greater than *18* , nothing will happen, the application terminates. This example has already started using *console I/O* knowledge .

A small note, is that in the case of the *if* block if there is only one line of code as in the above example, sometimes people remove both the { and } , ie the block of code defaults to only 1 line, then the above *if statement* would look like this.

```
if (age < 18)
```

```
System.out.printf("You can not access"); 2. if else
```

The syntax for the *if else statement* is as follows. if (condition_ expression) { commands_1;

```
} else {  
    command_2;  
}
```

The conditional *expression* will also return a *boolean* value . *commands_1* are executed in case the conditional *expression* is *true* . *command2* will be executed in case the conditional *expression* is *false* .

Example for *if else statement* .

```
if (age < 18) {  
  
    System.out.printf("You can not access"); } else {  
    System.out.printf("Welcome to our system!"); }
```

You see this example making it clearer that the user enters an *age* greater than *18* , then the conditional *expression* will return *false* , and so the *command_2* will be executed, in this example the line prints the console "Welcome to our. system! " .

3. if else if

The syntax for the *if else if statement* is as follows.

```
if (condition_expression_1) { commands_1;  
} else if (condition_expression2) { command_2;  
}  
...  
else {  
command_domains;  
}
```

This is a more extended form of the *if else statement*. Then *expression_1* returns a *boolean* value , and *statements_1* are executed in case *expression_1* is *true* .

If *conditional_expression_1* returns *false* , the system will check to *conditional_expression_2* , *command_line_2* be executed in case *conditional_expression_2* is *true* .

If the *expression* returns *false* , everything will be checked continue ... until any*conditional_expression* at all (not any guy returns *true*) then *command_line_n* will be executed.

Example for *if else if statement* .

```
Scanner scanner = new Scanner(System.in);  
System.out.println("Please enter a number of week (1 is Monday): "); int day =  
scanner.nextInt();  
  
if (day == 1) {  
  
System.out.printf("Monday"); } else if (day == 2) {  
System.out.printf("Tuesday"); } else if (day == 3) {  
System.out.printf("Wednesday"); } else if (day == 4) {  
System.out.printf("Thursday"); } else if (day == 5) {  
System.out.printf("Friday");  
} else if (day == 6) {  
System.out.printf("Saturday"); } else if (day == 7) {  
System.out.printf("Sunday");  
} else {  
System.out.printf("Invalid number!"); }
```

4. ?:

This statement is not really new, it is like an *if else statement* but is represented more concisely. This statement is also known as *the Tam Nguyen Operator*, because the two characters you see (? And :) help separate the components of the statement into three parts (or three operands), you can see the syntax of it is as follows.

[result =] conditional expression? command_if_true: command_atlate_false;

Which *result* may or may not, turn *result* will save the value as a result of the statement, it is the data type of *cau_lenh_neu_true* and *cau_lenh_neu_false*, why they invite you to read the next paragraph.

conditional_expression similar statements *if* above. *statement_if_true* will execute when the *conditional expression* returns *true*, this *clause* will return a result that can be of type *String*, *boolean*, *int*, ... Otherwise, the *if_false* when *conditional_expression* returns *false*, and this clause for *result*.

statement will execute will also return results

Example for ?: Statement (this example is rewritten from the *if else* example above).

```
Scanner scanner = new Scanner(System.in); System.out.println("Please enter your age: "); int age = scanner.nextInt();
```

```
String access = (age < 18) ? "You can not access" : "Welcome to our system!"; System.out.printf(access);
```

You also see that this statement is only suitable to replace *if else* only, and actually it helps us shorten the number of lines of code, but makes the algorithm harder to read, right. It's up to you to consider using *if else* or ?: Okay.

A little note is that with the above code, you do not need to use the *access result* variable, but print it directly to the console from this command, I adjust a little bit as follows.

```
Scanner scanner = new Scanner(System.in); System.out.println("Please enter your age: "); int age = scanner.nextInt();
```

```
System.out.printf((age < 18) ? "You can not access" : "Welcome to our
```

system!"); V. *Switch case statement*

This statement can be used to replace the *if else if* above if the conditional *expressions* all use the same object to compare (for example, in the *if else if* above we use the variable *day* to compare and compare against different values). At this point you should use *switch case* to make the *if else if* look more explicit.

The syntax for the *switch case statement* is as follows:

```
switch (comparison_object) { case value_1:  
commands_1;  
break;  
  
case value2:  
command_2;  
break;  
  
...  
default:  
command_domains;  
break;  
}
```

Instead of comparing the object in each *conditional_expression* as in the *if else if* you just need to transfer it to *comparison_object*, then assign each *value_x* of it to implement its results in *cac_cau_lenh_x* respectively.

You remember that each *case* ends with a special *break* statement (this *break* statement will be discussed in the next lesson). Currently, you should only know that this *break* statement helps you to stop execution at a block of certain *command_x*, if there is no *break*, the system will go through the next *case* to process.

The final component of this command is the like *else* last one *if else if*, it indicates that if the *VALUE* of *comparison_object*, then *command_line_n* in *default* will be call. keyword *default*, this component is

the comparison *case* did not meet

Example for *switch case statement* (this example is rewritten from *if else if* example above).

```
Scanner scanner = new Scanner(System.in);
System.out.println("Please enter a number of week (1 is Monday): ");
int day = scanner.nextInt();

switch (day) {
    case 1:
        System.out.printf("Monday"); break;
    case 2:
        System.out.printf("Tuesday"); break;
    case 3:
        System.out.printf("Wednesday"); break;
    case 4:
        System.out.printf("Thursday"); break;
    case 5:
        System.out.printf("Friday");
        break;
    case 6:
        System.out.printf("Saturday"); break;
    case 7:
        System.out.printf("Sunday");
        break;
    default:
        System.out.printf("Invalid number!"); break;
}
```

Do you see anything like *if else if* does.

We just went through *Conditional Commands Part of the Flow Control Statement*. Thereby you also understand the command block and variable scope in Java programs, please follow the next lessons to better understand this Java language.

Lesson 9: Loop Statement

Previous lesson we were acquainted with one of the *Statements Flow Control (Control Flow)*, which is the *conditional statement* (or *sentefnace branch instruction*). This we go on to the rest of that is the *Repeat Statements*.

I. Repeating Concept

Loop (in English called *Loop*) in programming is an act of repeating a block of statements when a certain condition is satisfied (also known as the result of that expression is *true*). If *Conditional Statements* help you branch off lines of code, then these *Repeat Statements* help you iterate over certain lines of code.

My example has a request that requires you to print to the screen *1000* numbers from *1* to *1000*, maybe you call *1000* times the command *System.out.println()*?

For a more practical example, if there is a request that you want to print out the names of all the students of your school (assuming you already know the command that reads a student information from the database), you might write. How does the code read each student's data and print it to the screen?

The examples above show us a clear practical concept and the necessity of using *Iterative Statements* in this's article.

II. Repeat Statements

We have 3 types of loop statements to clarify in this's lesson, they are: *while* , *do while* and *for* .

1. *while*

The syntax for the *while* statement is as follows. *while (condition) { commands;*

}

The syntax of the command *while* quite simple, initial program will test *condition* , if this condition returns the result *true* , then *COMMANDS* will be executed, and then the program will re-examine *condition* . Loop *while* only ended when *condition* returns the result *false* .

For example.

```
int i = 0;
while (i < 10) {
    System.out.println("Hello!");
    i++;
}
```

You see the above example to initialize the variable *i* (it is called a counter variable, because this is a variable used to control the number of iterations of the *while* loop). Starting in the *while*, you see the the inner *commands* are

condition_the *pair* is given that if variable *i* is less than 10, executed, in this example just the function prints the

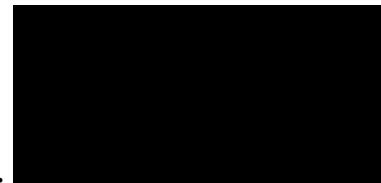
greeting "Hello!". Note one thing, in the *while* function body you always have to change the value of the counter variable, in this example *i* ++; increase count by 1 unit, so that to a certain time *condition* have been broken, this case *i* equals 10, the *while* function ends.

If you take the courage to remove line *i* ++; In the above example go and run again, you will see the print to the screen is called forever, people call this case endless loop.

- Practice # 1

Try applying the *while* loop to do the following: print the console the sum of even numbers from the sequence of integers of magnitude 1 to 10.

Note to check for an even number by doing the remainder division by 2, if the remaindering result is 0 then it is even.



Try the code, then compare with the following results.

Practice # 2

Use the *while* loop to find the prime numbers in the integer range 1 through 100 and print them to the *console*.

Note that prime numbers are numbers that are only divisible by 1 and themselves. For example numbers 2, 3, 5, 7, 11, 13, ...

Try the code first and compare it with the following results.

```
int number = 1;  
while (number <= 100) {  
    int count = 0;  
    int j = 1;  
    while (j <= number) {  
        if (number % j == 0) {  
            count++;  
        }  
    }
```

```
j++;  
}  
if (count == 2) {  
    System.out.println(number);  
}  
number++;  
}
```

2. *do while*

The syntax for the *do while* statement is as follows.

```
do {  
    commands;  
} while ( condition);
```

Have you noticed that the *do while* syntax is different from *while* in any way? That's if for *a while* , the system must check *condition* ago, if a new agreement is implemented *COMMANDS* .

As for *do while* , the system will execute the *commands* first and then check the *pair condition* to see if the repetition is needed.

So with *due while* the *COMMANDS* are executed at least one time, while the *while* the *COMMANDS* probably will not ever be done if *condition* not satisfactory.

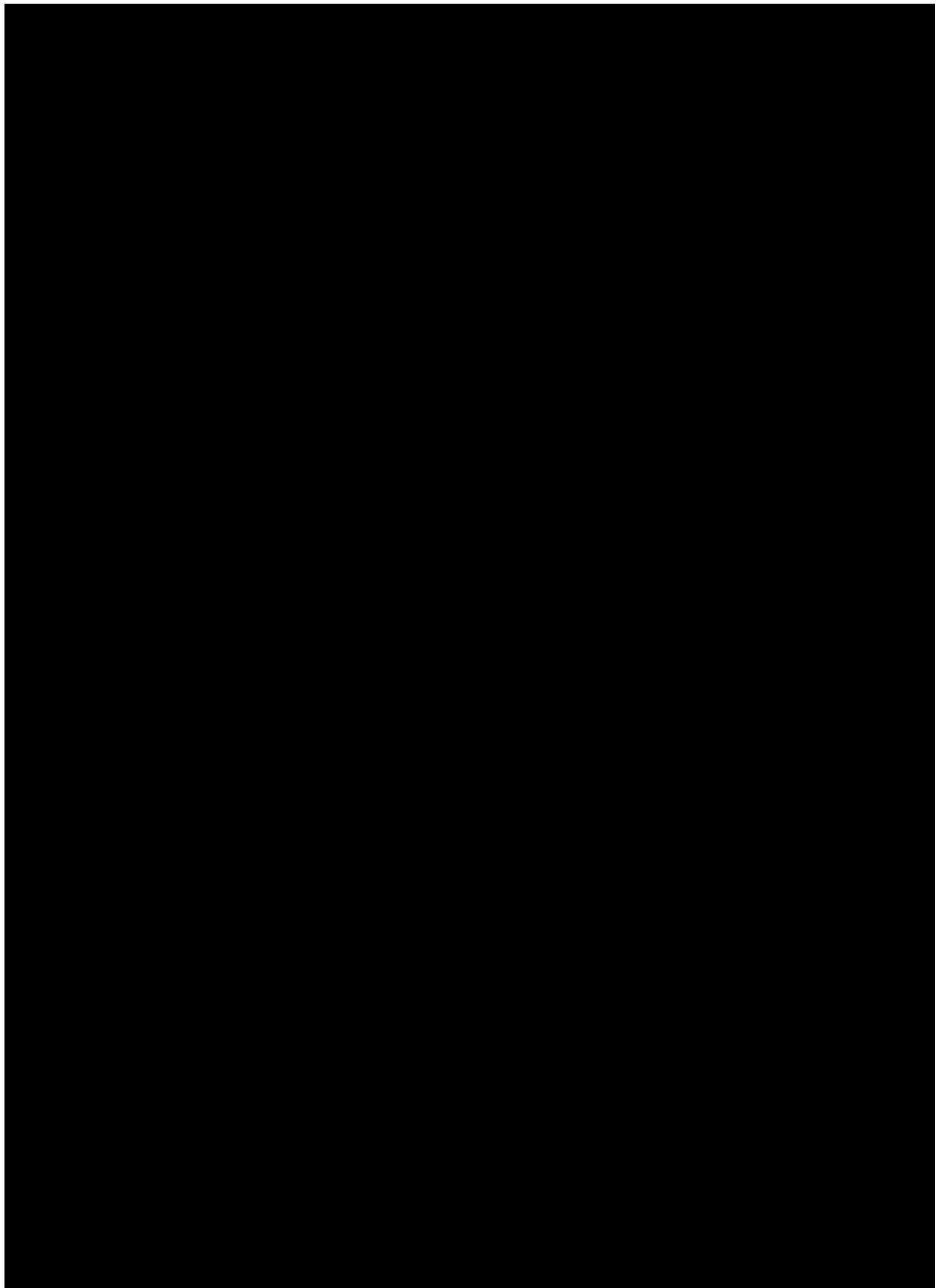
do while is less commonly used than *while* . You do not need to see examples for *do while* , let's start trying some exercises as follows.

- Exercise Number 3

Print out the console message asking the user to enter a number from the console, and then indicate the corresponding weekdays.

For 1 , print “Monday” ,... 7 print “Sunday” . The program will ask the user to enter the number repeatedly until they enter a number that is not a value between 1 and 7 .

Reference code is as follows.



3. *for*

The syntax for the *for* statement is as follows.

```
for (count_initiator; condition; count_conversion) {  
    commands;  
}
```

The *for* statement will loop *its inner commands* based on checking the three passed elements, separated by signs ; . Inside.

- *constructor_value* is the same as you declare a variable and initialize (assign it a value) that you learned in *lesson 4* . this *variable* will be the basis for the program to repeat your line of code how many times.

- *condition* the conditions set for loop checks whether to repeat to a round *COMMANDS* or not.

- *modifier* helps to change the value of the *count variable* every time *commands* are executed in one iteration, why? Because just like the loop above statements, without adjustment to the counter, the counter will never change the value, and *condition* will always return the same value, and so the loop will probably iterate indefinitely, or won't do any iterations.

You will understand the *for* statement more through the following example.

```
for (int i = 0; i < 10; i++) { System.out.println("Hello!"); }
```

The above example is very simple, the *variable initialisation* is the declaration and the initial value for variable *i* is *0* . the *pair_ condition* will repeat again and again when the variable *i* is less than *10* . *modifier* will increase *i* by *1* after execution of *commands* . the *commands* in this *for* loop are just the function that prints the console with the words "*Hello!*" . So think about how many "*Hello!*" Is the screen printed in the above example?

- Exercise Number 4

Please rewrite *exercise Number 1* on the other in the loop *for* offline.

```

int sumEven = 0;
for(int i = 1; i <= 10; i++) {
    if (i % 2 == 0)
        sumEven += i;
}

System.out.println("Sum: " + sumEven);

```

You can see that with *for* we can declare the counter variable *i* in the loop and initialize it in there, unlike *while* must be initialized outside. And the increase of variable *i* is to be in the third part of the *for* loop , not in the function body as with *while* .

- Exercise Number 5

Please rewrite *all Practice No. 2* on the other in the loop *for* offline.

```

for (int number = 1; number <= 100; number++) { int count = 0;
for(int j = 1; j <= number; j++) {

if (number % j == 0) {
count++;
}
}
if (count == 2) {
System.out.println(number);
}
}

```

- Expansion of for

Most of you will have trouble initially acquainted with *for* he, too, from the beginning known *for* until some time later his new spending good *for* , before I usually use *the while* instead *for* due *While* is easier to access and remember, but as you can

see *while* taking up more lines of code, each has its price
This expanded position for you are already knowledgeable about *for* now and want more *for* there is no other power slightly.

First, even though *for* is designed to pass in three elements, you can still miss one. For example, the code below *lacks* the count variable, then you have control over the counter variable inside the *for* function body .

```
for (int i = 1; i <= 10;) {  
    System.out.println("Hello!"); i++;  
}
```

Next, you can with the 2 part, and you have control of this defect through the controls inside and out *for* . Like the following example.

```
int i = 1;  
for (; i <= 10;) {  
  
    System.out.println("Hello!"); i++;  
}
```

More brutally, you can always miss 3 components, people often use this type of *for* loop to loop infinitely, used for some situations where it is not clear when to stop the loop, what conditions the loop will run. If you need to stop, there are functions "*jump*" out of the loop that we will be familiar with in the next article. Take a look at this infinite *for* example in the code below. And remember, don't try this endless for loop at home.

```
for (;;) {  
    System.out.println("Please don't try at home"); }
```

You have just walked through the rest of the *Flow Control Commands with you* , which are *Repeat Commands* . You will be told a little point regarding these statements, that the statements used to exit the loop while the loop condition remains, will be very useful in many practical cases.

Lesson 10: Break And Continue In The Loops

We will talk about two jump statements in this's lesson, which are *break* and *continue* . While there is another jump statement, *return* will be discussed in the following *Method* lesson .

The reason Java uses the word "*jump*" is not because it makes your loops more

jumpy. A jump is to jump out of the loop, or jump to the next iteration, ignoring the remaining statements within the loop body. Some other documents call this *the Control Commands*, but I prefer the word dance, Vietnamese sounds a bit banal, but in English they call it *Jump Statements*, it sounds better.

Let's take a look together.

I. break - Stop Statement

You understand that *the break was smashed* also true, but in this situation it means *stop*, the better.

As the name implies, when these statements appear somewhere in the loop, they break, or stop the loop, even though there are still other statements inside the unprocessed loop.

If you remember, in *lesson 8* we will talk through this *break* statement for the correct *switch case* structure. Well, *break* in *switch case* and *break* in loop statement have the same effect.

- Practice # 1

This exercise wants you to print out to the console all primes between 1 and 10,000.

Wait! Something's wrong! Printing prime numbers has been done in previous exercises! And printing prime numbers has nothing to do with *break* !?!

Rest assured, printing prime numbers in the previous lesson is just a chicken algorithm. In programming, especially algorithms programming, you will always welcome to have innovative algorithms so that the application runs quickly and smoothly. To do this, your algorithms must be neat, make the computer work less, the number of loops is reduced, ... Perhaps we will talk about this topic in another lesson.

Back in the exercise, we will improve the prime number finding algorithm, combined with the *break* statement to stop early checking once we know the number is not prime.

You already know prime numbers are numbers that are only divisible by 1 and themselves. To do this, in the old way you go through the terms 1 to itself, count how many terms it divides by, if you count 2 terms, it is a prime number. This way you should always let the loop run out all terms. Specifically, with the 10,000 check, your application will have to repeat 10,000 times to count. This way we measure the application takes 376 ms (milliseconds) to run, which is

almost half a second. You can apply a *break* to reducing the number of iterations in two ways.

- Do not let the loop run from 1 to itself, but let the loop run from 2 to itself reduced by 1 unit, ie run in interval [2, *itself*] . Whenever you find a number in this range that is divisible by itself, call *break* immediately to end the loop, no need to repeat because it is definitely not a prime number. With this algorithm is suppose to check number 10000 , your application needs to run to the class only two to have the break now. This takes 167 ms to print out all the primes, taking only half the time compared to the first. Oh so cool!
- This way is even better, according to research (not clear from any source), you just need to check in the range 2 to the square root of itself, that is for the loop to run in [2, *the square root of itself*] . If a number in this range is found that is divisible by, it is not a prime number. In Java, the function that takes the square root of a number is *Math.sqrt (real number)*; . This third algorithm only takes 15 ms to run, 25 times faster than the first.

I apply the fastest way above to the code below, how do you apply it? Let's try to code.

```
for (int number = 2; number <= 10000; number++) { boolean isPrime = true;
for(int j = 2; j <= Math.sqrt(number); j++) {

if (number % j == 0) {
// If appear only one j in this range
// number won't be a prime
isPrime = false;
break; // Get out of this for (the outside for is still running

}
}
if (isPrime) {

// if isPrime is still true, the number is a prime
System.out.println(number);
}
} II. continue - Skip Statement
```

Once again the meaning and use of this word have been confused. You speak

English should understand the meaning of *the continue* is *continuing* also not wrong, but we understand this situation is *ignored*.

Ignore means dropping the remaining statements inside the loop for a new loop. This command does not stop the loop as *break*, which only perform the new loop, the loop can be stopped by *the continue* when implementing new loop will check and see *dieu_kien_lap* no longer deal only.

There is a small note when you use *continue* for *for* and *while / do while* as follows. Because *continue* will execute a new loop on its own, there is a possibility that the counter will be ignored if you leave the statement incrementing after *continue*, so there will be a case of you falling into an infinite loop, this easy to see in *while* and *do while*. But for the peace of mind, statements *continue* in this loop involves increasing counter (if you have defined an increase in counter this third component of *for*) and then implement new loop should *continue* in *for* will be safer.

- Practice # 2

For this lesson, you'll have to print to the console as opposed to the exercise above, combined with *continue*. Write a program that prints to the console all numbers that are NON-PRINCIPAL numbers between 1 and 100.

We will apply the algorithm to find the prime number, knowing that it is a prime number, we will use *continue* to "flip" through the new iteration, the statements printed to the console will be placed under the *continue statement* to ensure printing. What is not eligible for *continue*. Our code is as follows.

```
for (int number = 1; number <= 100; number++) { if (number == 1) {  
    // 1 is not a prime  
    System.out.println(number);  
    continue;  
  
}  
boolean isPrime = true; for(int j = 2; j <= Math.sqrt(number); j++) { if (number  
% j == 0) {  
  
    //if we found one number is in this range, // number will not is a prime  
    isPrime = false;  
    break;  
}
```

```
}

}

if (isPrime) {
// if isPrime tell us this is a prime, we jump to next step // skip all the code
below continue
continue;

}

System.out.println(number); }
```

You have just walked through the rest of the statements that affect the iterative structure you have just learned.

Lesson 11: Array

This's lesson will focus on *Arrays* in Java.

First, I talk a little about *Arrays* before going into details. There are some Java documents or programs that talk very late about *Arrays* , often they spend talking about OOP before talking about *Arrays* . In my opinion, this makes sense, because in Java, *Array* is not a primitive data type, array is a new data structure based on the foundations of objectoriented, so, know about object-oriented. New to *Array* is not wrong at all.

However, to understand and use *Array* well in Java, I don't think you need a deep knowledge of OOP. Besides *Arrays* are very efficient, but if you use *Arrays* too late, I fear you will lose some good opportunities to gain access to some interesting knowledge of Java. Furthermore *Arrays* can also be seen as the foundation for creating the concept of *string* (*String*) that we last mentioned at all after that, again *String* is also a concept of OOP, but we also have to get used soon .

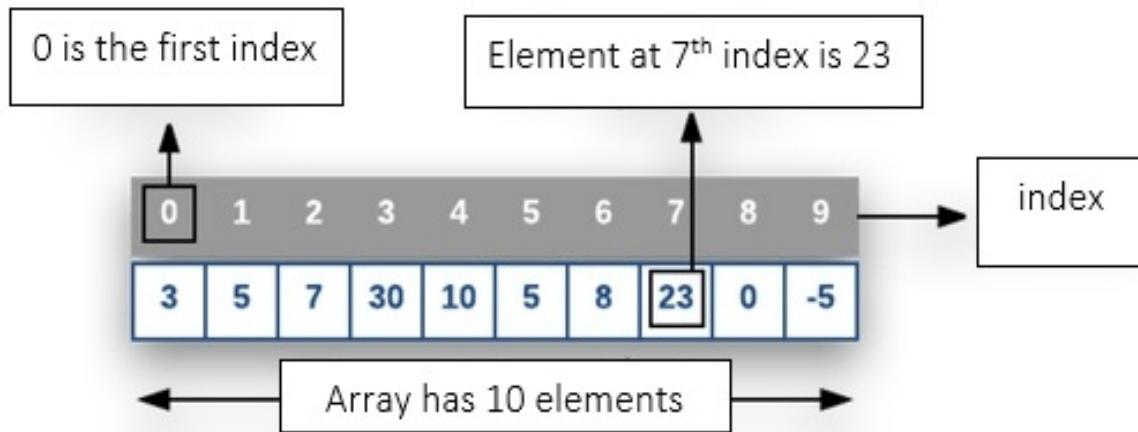
I. Array Concept

An *array* , also known as an *Array* , is a data structure used to contain a set of elements with similar data types. Assuming we have an *array of integers* , this array will contain a set of elements of the same data type *int* . In addition to the set of integers as the example just mentioned, or the set of primitive data types that you are also familiar with, the *Array* also contains a set of "*non-primitive*"

data that we will talk about in the following examples.next lesson more.

When you declare an array, you must *specify how large* it is, that is , *specify the maximum number of elements* the array can contain. Each element in the array will be managed according to the *Index* (*Index*), the index will start with number 0 (you will be familiar with how to access the array elements based on the indicators in the example below) .

You do not need to know much, please relax, let your mind gently approach the *Array* . Suppose we want to create an *Array with up to 10 elements of type int* , and when storing this array in memory, they will be arranged together according to the following mock-up model, each cell is an element of type *int* with number of which is the value stored in.



You already know more about *Arrays* , so why use this structure?

II. Why Use Arrays?

The meaning of this section is to talk about the benefits of using *Arrays* .

True to the name and definition of the *Array* above, the *Array* data structure will help you to store the list of elements, for example, you will need to store a list of students. This list will be organized so that you have *quick random access* to each child element, for example you want to get student information at position 10. Boom! Right now!. This list can be rearranged according to one criterion, so you can quickly read out the top 10 students with the highest scores. Or another advantage of *Arrays* is that the organization of elements with similar data types

will make our code more transparent and manageable, helping to optimize the code.

However, *Arrays* also have a disadvantage that you must declare the magnitude, ie, predeclare the number of elements that this *Array* will use. This makes your code less flexible because sometimes we want an array with more elements, or reduce the performance of the system because you pre-declare an array of too high size without using up. the number of devices declared. Of course, to overcome this disadvantage of *Arrays*, Java also has the concept of *ArrayList*, which we will become familiar with in another lesson.

III. Array Usage

Just like when you are familiar with the concept of variables, you must know how to declare, assign data, and use that variable in an expression. With *Arrays* too, but because *Arrays* bit special is that it contains child elements, so beyond what you can apply from the use of variables, there will be a few new things here which you must be familiar . Let's go through the steps in using *Arrays* as follows.

1. Array Declaration

You can choose one of the following two declaration methods.

`data_type [] array name;`

or

`array name_data [];`

Remember, there must be a `[]` pair in the *data type* or *array name* . The reason Java defines both methods as above is because the first is true to Java standards, while the second way, Java supports the declaration of *Arrays* from C / C ++.

The `[]` pair also shows that this is an array, you noticed that if there are no brackets, the above declaration is no different from declaring a variable right. Uhm ... But unlike variables in that the *Array* in this declaration step is still not usable, as said, we need to specify the size of the array, so let's go through the next step. Now an example. To declare an array of integers, we code as follows.

```
int[] myArray;
```

2. Allocate Memory For Arrays

Again you must use the *new* keyword to allocate memory. The last time you used

new was in *Lesson 7* , when you initialized a *scanner* variable , do you remember.

```
Scanner scanner = new Scanner(System.in);
```

With *Arrays* , the memory allocation would be similar, as follows.

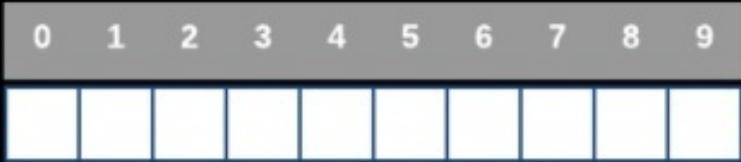
```
array name = new data_type [array size];
```

In this step it is necessary to have the *array size* . If not passed this parameter, the allocation will generate an error. This size is at your discretion, normally in an application, if you know this size parameter exactly and this size will not be changed during the life of the application, then use an array, If we do not know exactly or the size is always changing, we should use *ArrayList* as we will discuss later.

Following the declared example above, we add the allocation as follows.

```
int[] myArray = new int[10];
```

When the application runs to this step, it will create an array with 10 "empty" integer elements as shown below.



3. Array Initialization

As you can see, the above allocation step is just like a placeholder in the system of a memory area large enough to hold 10 integer elements. The next thing for the array to function is to initialize, also known as storing data in those empty cells.

There are many ways to initialize an array.

a. The first way

Initialize as soon as allocating memory for the array, then you do not need to use the keyword *new* anymore, just declare and initialize on one line as in the example below. Use this method when you know in advance how you need to initialize an array. With the example array of type *int* as above, I will initialize the array in the first way as follows.

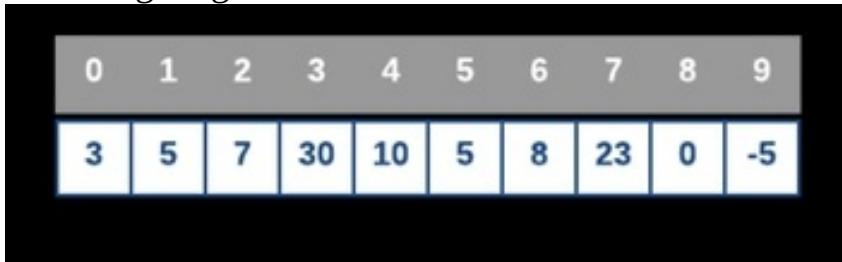
```
int[] myArray = {3, 5, 7, 30, 10, 5, 8, 23, 0, -5};
```

b. The Second Way

Initialize each value for an array by retrieving them based on an index and assigning them a value. This is used when you don't know how to initialize the array in the first place, but then, with the logic of the application, you will sequentially populate each element according to its index.

```
int[] myArray = new int[10]; myArray[0] = 3; myArray[1] = 5; myArray[2] = 7;  
myArray[3] = 30; myArray[4] = 10; myArray[5] = 5; myArray[6] = 8;  
myArray[7] = 23; myArray[8] = 0; myArray[9] = -5;
```

Both of the two ways above produce an array in the system expressed as the following diagram.

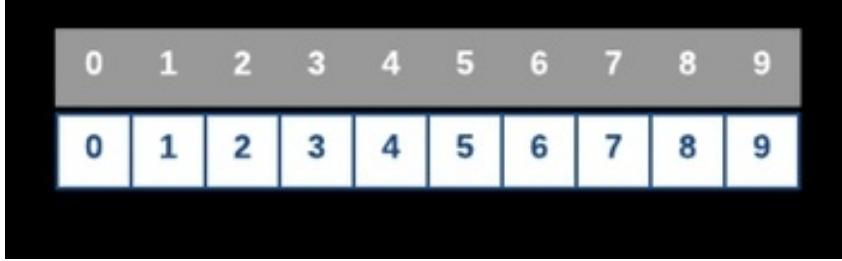


c. Different ways

In addition, sometimes you also use the loop to initialize values, in case the array values are the same, or in a certain order, our example with initialization is as follows.

```
int arraySize = 10;  
int[] myArray = new int[arraySize]; for (int i = 0; i < arraySize; i++) {  
    myArray[i] = i; }
```

Then the array will contain a list of values like this.



4. Access Arrays

As you are familiar with the ways above, we will *access arrays based on indexes*. You should always remember that the first index (index number) of an array

element starts at 0 , and that the index of the nth element will be n-1.

, and that the index of the nth element will be n-1.

element array.

```
int[] myArray = {3, 5, 7, 30, 10, 5, 8, 23, 0, -5};  
System.out.println("10th element in array is: " + myArray[9]);  
Or if you want to print out all element values in the array, the best way is to use  
the for loop.
```

```
int[] myArray = {3, 5, 7, 30, 10, 5, 8, 23, 0, -5};  
for (int i = 0; i < myArray.length; i++) {  
    System.out.println("Element in " + i + ": " + myArray[i]);  
}
```

Note that if you use *myArray.length* , the system will return the size of the array, in this example the magnitude is 10. Operator *.**.* in the above expression will be more detailed in the OOP section.

IV. Some Practice With Arrays

- Practice # 1

Create an array of integers and initialize the array as {3, 5, 7, 30, 10, 5, 8, 23, 0, -5} . Print out the TOTAL and ADDITIONER console of the element values in the array. And here is the code for the exercise.

```
int[] myArray = { 3, 5, 7, 30, 10, 5, 8, 23, 0, -5 };  
int sum = 0;  
double avg;  
int count = myArray.length;  
for (int i = 0; i < count; i++) {  
  
    sum += myArray[i];  
}  
avg = (double) sum / count;  
System.out.println("Sum is " + sum);  
System.out.println("Avegare is " + avg);
```

- Practice # 2

With an array of integers as in the above exercise. Print the POSITION (the order) of the elements less than or equal to 0.

And the code of the program.

```
int[] myArray = { 3, 5, 7, 30, 10, 5, 8, 23, 0, -5 };
int count = myArray.length;
boolean isFound = false; for(int i = 0; i < count; i++) {

if (myArray[i] <= 0) { System.out.println("The position below or equal zero is: "
+ i); isFound = true;

} }
if (!isFound) { System.out.println("Can not found the position you need");
• Exercise Number 3
```

Also with the array of integers as above exercise. Now rearrange the test arrays in ascending order, so that when printing to the console the content will look like this "-5 0 3 5 5 7 8 10 23 30" .

And the code is as follows.

```
int[] myArray = { 3, 5, 7, 30, 10, 5, 8, 23, 0, -5 };

for (int i = 0; i < myArray.length - 1; i++) { for (int j = i; j <= myArray.length - 1; j++) { if (myArray[i] > myArray[j]) { // Thao tác này đổi 2 giá trị ở 2 vị trí i, j của mảng int temp; temp = myArray[i]; myArray[i] = myArray[j];
myArray[j] = temp; }

}
for (int i = 0; i < myArray.length; i++) { System.out.print(myArray[i] + " "); }
```

Together we go through another new knowledge in Java, knowledge of *Arrays* . However, the knowledge of *Array* mentioned in this's lesson is still a little bit more, not to mention because the lesson is also quite long, and... too long, you are lazy to read :). That said, the total knowledge of *Arrays* is not much and difficult, this amount is enough for you to experiment with solving problems on the internet or in class, the following section just adds how to use *foreach* on arrays, and a little difficult problem that is a multi-dimensional array. Keep on learning!!

Lesson 12: Array (Continue)

Remember *the previous* lesson, we talked about the concept and how to use the Array in Java, then we also mentioned *the most effective way to traverse the elements in the Array is to use a for loop*. This we learn another "variant" of for for Arrays, which helps you to browse Arrays faster than what we said together in the lesson for and about Arrays, which calls is *foreach*. Another enhancement to Arrays will be discussed in this's lesson, which is *Multi-Dimensional Arrays*.

I. **foreach**

1. *Foreach Concept Through Questions*

First question: why don't I talk about foreach in the foreach *lesson*? The answer is, it is true that foreach is also for ... because it is a combination of for and each ... the syntax of foreach also inherits from for, and the way of foreach behavior will of course be the same as for. But foreach was born to work with Arrays, foreach helps implement lines of code to navigate the array easier. That is why foreach is always included with the concept of Arrays.

The next question is: is foreach important? The answer is, it doesn't matter so you won't have to use foreach. As mentioned in the above answer, foreach only makes the implementation of code on the loop easier, you can completely use traditional for to traverse the Array without the foreach.

2. *How to Use foreach*

Let's talk about the foreach syntax first.

```
for (duplicate_character: array) {  
    commands;  
}
```

Do you see the same thing as for? For for, you need to include 3 components to define its repetition rules. As for foreach, obviously this loop will automatically know it must go through the elements in the array in turn, so the parameter for this loop becomes much simpler than for. Invite you to familiarize yourself with each element in this foreach syntax.

- *declare_partners* is where you will declare a new variable to use in the body of this foreach function. This variable will contain the value of the element that this loop is iterating on, so it must have the same data type as each array element.

- *Array* is the Array you need to loop. This can be a function that returns an Array value. Functions, also known as Methods, will be discussed in *the next*

lesson .

- *Colon (:) is understood as "in"* , so the meaning for all parameters in this *foreach* is "*each element in the array*" .
- the *command_commands* are where you use the variable in the *escape_command declaration* .

3. Practice # 1

Let's go back to practice # 1 in Array the other day and try again *foreach* in this's lesson. I copied the practice content from the previous day as follows.

Create an array of integers and initialize the array as {3, 5, 7, 30, 10, 5, 8, 23, 0, -5} . Print out the TOTAL and ADDITIONER console of the element values in the array. And here is the code for the exercise.

```
int[] myArray = { 3, 5, 7, 30, 10, 5, 8, 23, 0, -5 };
int sum = 0;
double avg;
for (int i : myArray) {

    sum += i;
}
avg = (double) sum / myArray.length;
System.out.println("Sum is " + sum);
System.out.println("Avegare is " + avg);
```

4. When Should You Not Use *foreach*?

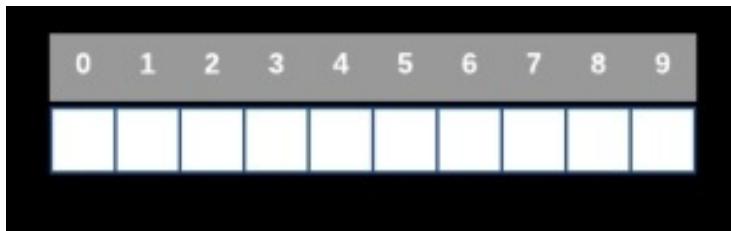
Although *foreach* is pretty cool, I love it too, but remember that you can't always use *foreach*, and here are the exceptions.

- Do not use *foreach* to remove an element from the List (I use the word "*List*" and not "*Array*" , because Arrays do not allow you to add or remove an element in it, only List is new. allows, and because *foreach* will also work on Lists similar to Arrays, so I mentioned this point here, later on, I will remind you of Lists). Since *foreach* is sequentially traversed on array elements, it's bad at determining the position (index) of each element. So don't force the *foreach* to work when you want quick access to the location of any element.

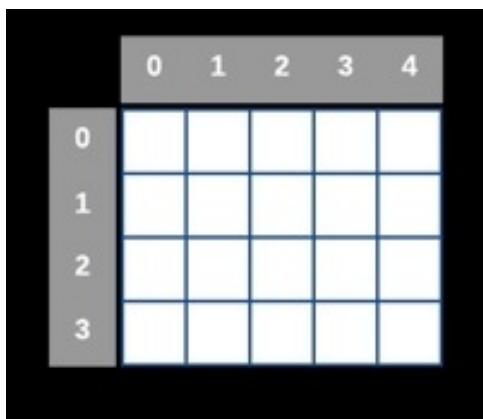
II. Two-dimensional array

By this step, you should definitely understand the concept of Arrays. When

people talk about Arrays, they mean "*One-dimensional Arrays*" , as you know this Array is treated as a (fixed) list of elements stretched in a single dimension. As illustrated in the previous lesson, the Array (one dimension) looks like this.



What do you think if there is a new one-dimensional Array representation, then your Array is called a *two-dimensional Array* , which is also called *Matrix* . The representation of this Two-Dimensional Array looks like this.



1. Declare Two-Dimensional Arrays

Similar to the Array declaration, declaring a two-dimensional Array is similar, but requires two square brackets [] [].

data_type [] [] array name;

or

array name_data [] [] ; We try to declare Two-dimensional Array for the above illustration.

int[][] myMatrix;

2. Allocate Memory For Two-Dimensional Arrays

As well as Arrays. The syntax for allocating a two-dimensional Array is as follows. array name = new data_type [line_number] [column_number];

So with the declared example for the two-dimensional array above, we progress the allocation as follows.

```
int[][] myMatrix = new int[4][5];
```

3. *Creating Two-Dimensional Arrays* Again like Arrays, Two-dimensional Arrays also have many initialization forms. Let's say you want to create a two-dimensional Array with the following values.

		0	1	2	3	4
0	3	5	7	30	10	
1	5	8	23	0	-5	
2	100	-9	4	2	55	
3	-80	-22	11	1	12	

a. The first way

Use when you know in advance the data to be initialized for a two-dimensional Array, like the data you gave above, you initialize it as follows.

```
int[][] myMatrix = {
```

```
{3, 5, 7, 30, 10}, {5, 8, 23, 0, -5}, {100, -9, 4, 2, 55}, {-80, -22, 11, 1, 12}};
```

b. The Second Way

Use this way when you do not know the original data, depending on the program logic, the two-dimensional array will be populated slowly, then you have to work with its index. One note is that with a two-dimensional Array its index is shown as follows.

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]
[3][0]	[3][1]	[3][2]	[3][3]	[3][4]

And to work with the index, you should pay attention to how to use the index as declared later.

```
int[][] myMatrix = new int[4][5];
```

```
myMatrix[0][0] = 3; myMatrix[0][1] = 5; myMatrix[0][2] = 7; myMatrix[0][3] =  
30; myMatrix[0][4] = 10; myMatrix[1][0] = 5; myMatrix[1][1] = 8; myMatrix[1]  
[2] = 23; myMatrix[1][3] = 0; myMatrix[1][4] = -5;
```

```
myMatrix[2][0] = 100; myMatrix[2][1] = -9; myMatrix[2][2] = 4; myMatrix[2]  
[3] = 2; myMatrix[2][4] = 55;
```

```
myMatrix[3][0] = -80; myMatrix[3][1] = -22; myMatrix[3][2] = 11;  
myMatrix[3][3] = 1; myMatrix[3][4] = 12;
```

c. Different ways

Just like Arrays, Two-dimensional Arrays can also be initialized with a for loop, of course by *nesting two for together*. Use this way for initializing values for the Two-dimensional Arrays in a certain order. See the following example.

```
int row = 4; int column = 5; int[][] myMatrix = new int[row][column]; for (int i  
= 0; i < row; i++) { for (int j = 0; j < column; j++) { myMatrix[i][j] = i + j; } }
```

Do you know how the above code will generate a Matrix. The Matrix you created is the illustration below.

		0	1	2	3	4
		0	1	2	3	4
0		1	2	3	4	5
1		2	3	4	5	6
2		3	4	5	6	7

Surely you have understood the two-dimensional array or the matrix well, but if you have any questions, please leave a comment below this lesson for yourself. Let us now go to practice for the Matrix.

4. Practice # 2

In this lesson we will try to manipulate Matrix. But instead of declaring and pre-initializing a Matrix, try importing them from the console, it will be interesting, the content of this exercise is as follows.

Create a Matrix of integers by.

- Print out the console the line "*Please enter number of row:*" and wait for the user to enter the number of rows of the Matrix.
- Print out the console the line "*Please enter number of columns:*" and wait for the user to enter the number of columns of the Matrix.
- Print out to the console line by line asking the user to enter each element of the Matrix. With the User-entered Matrix above, do the following.
 - Print out the matrix that the user just entered.
 - Print out the row and column with the largest sum in the Matrix.

If you understand the requirements of the practice, then proceed to code, code is complete, then invite you to compare with your results below. Welcome to share your thoughts or algorithms on this exercise in the sections below.

The sort of program when running, and when interacting with, the user looks like this.

```
Please enter number of row: 3
Please enter number of column: 4
Matrix[0][0] = 1
Matrix[0][1] = 4
Matrix[0][2] = -1
Matrix[0][3] = 9
Matrix[1][0] = 2
Matrix[1][1] = 5
Matrix[1][2] = 3
Matrix[1][3] = -10
Matrix[2][0] = 2
Matrix[2][1] = 4
Matrix[2][2] = 9
Matrix[2][3] = 3
Your Matix here
1 4 -1 9
2 5 3 -10
2 4 9 3
Max row in Matrix is row: 2
Max column in Matrix is column: 1
```

And the code is as follows.

```
Scanner scanner = new Scanner(System.in);
// Enter number of row System.out.print("Please enter number of row: ");
int row = scanner.nextInt();

// Enter number of column System.out.print("Please enter number of column: ");
int column = scanner.nextInt();

int[][] myMatrix = new int[row][column];
// Tell user to enter values of elements in matrix for (int i = 0; i < row; i++) { for
(int j = 0; j < column; j++) { System.out.print("Matrix[" + i + "][" + j + "] = ");
myMatrix[i][j] = scanner.nextInt(); } }

// Print matrix System.out.println("Your Matix here");
for (int i = 0; i < row; i++)
{
```

```

for (int j = 0; j < column; j++) {
    System.out.print(myMatrix[i][j] + " ");
} System.out.println(); // Line down

}

// Find max row int[] sumRow = new int[row]; // Creat array save the sum of
each row for (int ro = 0; ro < row; ro++) {

for (int co = 0; co < column; co++) { sumRow[ro] += myMatrix[ro][co];
} } int maxIndexRow = 0; int maxSumRow = sumRow[maxIndexRow]; //
Suppose that sum of row 0th is biggest for (int i = 1; i < row; i++) {
if (maxSumRow < sumRow[i]) { maxSumRow = sumRow[i]; maxIndexRow =
i; // Save the index of the max row
} } System.out.println("Max row in Matrix is row: " + maxIndexRow);

// Find max column int[] sumColumn = new int[column]; for (int ro = 0; ro <
row; ro++) { for (int co = 0; co < column; co++) {

sumColumn[co] += myMatrix[ro][co]; } } int maxIndexColumn = 0; int
maxSumColumn = sumColumn[maxIndexColumn]; for (int i = 1; i < column;
i++) {

if (maxSumColumn < sumColumn[i]) { maxSumColumn = sumColumn[i];
maxIndexColumn = i;
} } System.out.println("Max column in Matrix is column: " +
maxIndexColumn);

```

III. Is There Any More Array Types? !!

For the rest of the beard part of this's lesson, I would like to put all of it here, I just talk through it at a time without practice, nor do you need to understand, because this section talks about the remaining types of arrays in Java that most of us will not use. The truth is I have never used these types of "*flag*" arrays , but maybe somewhere you will encounter it, then let's take a look at what it is.

1. Three-Dimensional Arrays

Yes Two-dimensional array is enough to make you dizzy, now there is the

category *Threedimensional array* , even 4-dimensional, ... but wait ... you don't need to use too manydimensional array, let's try with array 3 In the afternoon, the operation on the array will be as follows, enough to discourage you already.

```
int[][][] array3D = new int[4][5][3];
for (int row = 0; row < 4; row++) { for (int col = 0; col < 5; col++) {
    for (int ver = 0; ver < 3; ver++) { array3D[row][col][ver] = row + col + ver;
} } }
```

2. *Serrated Plate* With this type of array, I stay away. Basically this array is represented "*irregularly*" , for example you have a Matrix with m rows, each row has a different number of columns.

```
int[][] myJaggedArr = { { 3, 4, 5 }, { 77, 50 } }; for (int i = 0; i <
myJaggedArr.length; i++) { for (int j = 0; j < myJaggedArr[i].length; j++) {
System.out.print(myJaggedArr[i][j] + " "); } System.out.println();
}
```

Finally, the knowledge related to Arrays is finished, hope you have a solid and good knowledge to apply in the following Java lessons

Lesson 13: String

In the previous lessons, we devoted all of our energy to talk and learn about Arrays. You also know that Arrays are a powerful and efficient data structure, they help manage the list of elements with the same data type, and also provide very fast access to any element. This, you are familiar with another pretty good application of Arrays, which is *Array of characters* , or called with a shorter and easier name, which is *String* .

I. Chain Concept

String , also known as *String* is simply an *Array of characters* . However, basically that is, but in terms of structure, *Chain is an Object* (this new concept we will talk about in *Object Oriented - OOP*). And since it is an Object, it will be systematically built inuseful *Methods* , our job is to get familiar with these Methods and get used when needed, such as string comparison method, chain cut , replace, find length, find substring....

When you get acquainted with Strings, you will hear somewhere that Strings are *immutable*, in English called *Immutable*. You can understand that once you initialize a String, the initialization value will be fixed. Every time you change a value for a String, the system creates a new Chain. So it won't matter much if your program creates Strings for use without you having the need to edit the Sequences or just a little like the exercises in this's lesson. But if you have a String that is constantly being changed, then instead of declaring a String of type String, you can use *StringBuffer* or *StringBuilder*. replace. This's lesson will not be enough to talk about StringBuffer and StringBuilder, I will focus on String and see you continue in the next lesson.

II. Declare And Initialize A Chain

I would like to test you a bit. Based on all that I said above, suppose now I ask you *to try declaring a String and initializing that String with the content "Hello World!"*, you will immediately think of the following declaration, right?

```
char[] chuoi = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'};
```

Not true, the above code just helps you to initialize an Array of characters, with String is much easier, you can choose one of two ways to declare and initialize as follows. *Initialized as "primitive"*, that is, you treat String as a variable with primitive data type. I give the declaration example for the string "*Hello World!*" always, but do not include the following syntax.

```
String myLiteralStr = "Hello World!";
```

Or initialize as "object", that is, you use the *new* keyword to initialize as what you did with the *Scanner* or *Array* in the previous lesson, you see.

```
String myObjectStr = new String("Hello World!");
```

You pay attention in the declaration and manipulation of String we enclose the string in double quotes (""), not single quotes (") as with Character.

Talk about a little knowledge, the two initializing sequence above two are actually different, because related to nature *can not be changed* is that I have mentioned above. But I'm not going to talk about those differences in depth here. Remember when I first approached Java, I only remembered the two above initialization methods, and I like the first one most, which is intuitive and very fast. I will set up a separate section to discuss these two initializations in another lesson.

III. Some Useful Chain Methods

At this point, I do not know what style of presentation is the best. Since Chain is still an Object Oriented (OOP) concept in principle, it will be a bit distant to most of you who are new to programming. But I think it's okay, using String is quite easy, I will try to list all the useful methods of String, please familiarize yourself with how to use these methods, then if you ever meet again Somewhere you should remember this's lesson.

1. String Comparison

When you have two strings, and want to know whether they are the same or different, or how they differ, the following comparison functions can be used.

a. equals ()

If there are two strings, for example, *chuoi1* and *chuoi2* , you can call *chuoi1.equals(chuoi2)* , the result is a *boolean* with *true* being the same and *false* is not the same.

b. Practice # 1 With the original string "*Hello World!*", Print the result of comparing the original string against the two strings "*HelloWorld!*" and "*hello world!*" Please.

```
String rootStr = "Hello World!";
System.out.println("Compare 1: " + rootStr.equals("HelloWorld!"));
System.out.println("Compare 2: " + rootStr.equals("hello world!"));
```

The results of the two commands printed to the console above are obviously *false* and *false* .

c. equalsIgnoreCase ()

The usage of this method is the same as *equals ()* , but *equalsIgnoreCase ()* returns the result of the comparison without *case sensitive* . That is, String "*A*" and "*a*" will be the same in this comparison.

d. Practice # 2

Please print out the result comparing the string "*Hello World!*" and "*hello world!*" Please.

```
String rootStr = "Hello World!";
System.out.println("Compare: " + rootStr.equalsIgnoreCase("hello world!"));
The result of the comparison statement is certainly true .
```

e. Comparison With Operator ==

At the level of this's lesson, we are *not talking about using the == operator* to compare two Strings in Java. Grammatically correct, you can try using this operator to practice comparing *chuoi1 == chuoi2* . But once you do not understand String and Object Oriented, I advise you to leave your mind of comparing two Strings in Java with == operator , I will say why in another lesson.

f. compareTo ()

This comparison method is quite special, it will take each character to compare, for each character it will get *the Unicode value* and then subtract these characters, once we have detected a different character. (subtraction yields a negative or a positive result) the subtraction stops.

Specifically, for example *chuoi1.compareTo (chuoi2)* then.

- *The result is a negative number* when *chuoi1* has less than *chuoi2* character order in Unicode .
- *The result is 0* when *chuoi1* is
- *The result is a positive number* when *chuoi1* has higher than *chuoi2* .

identical to *chuoi2* . Unicode character order

g. Exercise Number 3

Suppose you get somewhere two strings "1.5.5.3" and "1.5.6.1" , these are the two versions of the application. How can you print out which console version is the latest version?

```
String version1 = "1.5.5.3";
String version2 = "1.5.6.1";
int compare = version1.compareTo(version2); if (compare < 0) {

System.out.println(version2 + " mới hơn " + version1); } else if (compare > 0) {
System.out.println(version1 + " mới hơn " + version2); } else {
System.out.println(version1 + " giống " + version2); }
```

2. Chain Joins

When you have two strings, and want to join them together, you can use the following functions.

a. Matching String With Operator +

It's as easy as $1 + 1 = 2$ so let's look at the following example.

```
String str1 = "Hello World!";
```

```
String str2 = str1 + " Hello TechJA!";
```

`System.out.println(str2);` With this method of concatenating a String with the + operator , you can freely concatenate as many Strings together. And remember, there are no operators , * or / with String.

With the above example of String concatenation with operator + above, remember, you are familiar with how to join this String from the previous lessons and even this, in the exercises above, Those are functions that print to the console, please verify it.

b. concat ()

With the call `chuoi1.concat(chuoi2)` the result will be a new String similar to calling `chuoi1 + chuoi2` above.

c. Exercise Number 4

Please rewrite the statement concatenated with operator + above with `concat()` function .

```
String str1 = "Hello World!";
```

```
System.out.println(str1.concat(" Hello TechJA!"));
```

3. Child Chain Extract

If you have a String, and want to get a Sub-String from this Original String, please familiarize yourself with the following two functions.

a. subString (int startIndex)

If you call `chuoi.subString (startIndex)` then the result will return a substring with the content starting at `startIndex` of the `string` , note that as with Array, the *index of the characters in the string is started from 0* .

b. Exercise Number 5

With the original string "`Hello World! Hello TechJA`" Extract the substring "`Hello TechJA`" Please.

```
String rootStr= "Hello World! Hello TechJA";
```

```
System.out.println(rootStr.substring(13));
```

c. subString (int startIndex, int endIndex) Same as above, but if you call `chuoi.subString (startIndex, endIndex)` the result will return a substring with

content starting from *startIndex* to *endIndex* of the *string* .

d. Exercise Number 6

With the original string "*Hello World! Hello TechJA*" , please try to extract the substring as "*TechJA*" .

```
String rootStr= "Hello World! Hello TechJA";
```

```
System.out.println(rootStr.substring(13, rootStr.length() - 2));
```

Note in this exercise that I call the *length ()* function of the String, which will return the size of the String, the total number of characters in that String.

4. Convert In Capital - Normal Print

The String section has the following functions so you can convert uppercase or lowercase to all characters in String.

a. *toUpperCase ()*

If you call *chuoi.toUpperCase ()* then it will return a new String with all the characters capitalized from the *string* .

b. Practice # 7

With the original string "*Hello World! Hello TechJA* " Please capitalize all the characters of the string and print it to the console.

```
String rootStr= "Hello World! Hello TechJA"; rootStr = rootStr.toUpperCase();  
System.out.println(rootStr);
```

c. *toLowerCase ()*

This method is the exact opposite of *toUpperCase ()* , which will return a String with all the characters in the *string* to lowercase.

5. Some Other Common Chain Methods

Here are some other common String methods that you need to be aware of. a.

trim ()

This method removes spaces before and after the String.

b. Exercise Number 8

With the original string "*Hello World!* " . Cut off leading and trailing spaces and print them to the console.

```
String rootStr= " Hello World! ";  
rootStr = rootStr.trim();  
System.out.println(rootStr);
```

c. *startsWith ()* And *endsWith ()*

These two methods both return a type of *boolean* , indicating whether the String

begins or ends with a compared String.

d. Exercise No. 9

With the original string "*Hello World! Hello TechJA*" , print out the console whether this string starts with "*Hello*" and ends with "*Hello*" or not.

```
String rootStr= "Hello World! Hello TechJA";
System.out.println("String start with Hello? " + rootStr.startsWith("Hello"));
System.out.println("String end with Hello? " + rootStr.endsWith("Hello"));
```

The results of the two statements printed to the console above are *true* and *false* respectively .

e. `charAt()`

This method returns the character of the String at the passed index.

f. Exercise Number 10

With the original string "*Hello World! Hello TechJA*" Print the console with the 11th character in the string.

```
String rootStr= "Hello World! Hello TechJA";
```

```
System.out.println("The 11th char is " + rootStr.charAt(11));
```

This method has two parameters passed are two String, like this *chuoi.replace(chuoi1, chuoi2)* . If *string1* exists in *chuoi* then all the places that appear *string1* in *chuoi* will be replaced by *string2* .

h. Exercise No. 11

With the original string "*Hello World! Hello TechJA*" Please replace all "*Hello*" in this original string with "*Hi*" .

```
String rootStr= "Hello World! Hello TechJA"; rootStr = rootStr.replace("Hello",
"Hi"); System.out.println(rootStr);
```

i. `indexOf()`

This function will look in the String for the position of the first occurrence of the character passed in the function.

j. Exercise No. 12

With the original string "*Hello World! Hello TechJA*" Print to the console the position of the character "!" .

```
String rootStr= "Hello World! Hello TechJA";
```

```
System.out.println("The ! char at " + rootStr.indexOf("!"));
```

The output of this command will print out console number 11.

In addition to the methods listed above, the String has quite a few other useful methods, but in the framework of this lesson, I temporarily do not finish. Or there are more Object Oriented methods that can be confusing, so when I use these special methods, I will speak more clearly.

So you have a quick look at String in Java. Personally, I do not see enough knowledge about the String, there are still many interesting things, maybe I will create a new lesson to say more clearly.

Lesson 14: String Buffer and String Builder

This lesson is an extension of the previous meal's *series series*. It can be said that it is a supplementary lesson, it is not very important, but if not speaking, I feel guilty, and restless. If you care about the *performance* of the application, then pay close attention to the lessons of this type.

To get familiar with the knowledge and examples of this's lesson, you need to practice a little with the functions of the previous series to understand the basic of this object, then this you can get acquainted. it is easier with the concepts and methods of extension.

I. Why Know StringBuffer And StringBuilder?

Of course, this will be your first question when I introduce these two subjects this.

Well... actually, this's lesson I also considered a lot. You also know that when you learn about the Chain, it is considered that you have "*put one foot*" on the door of OOP, but we do not have any lessons about OOP. However, StringBuffer and StringBuilder of this's lesson continue to talk about OOP. But I think it's not good if I know about them too late after I finish OOP. So I decided to say it first, in the future, getting acquainted with OOP will surely understand this's lesson better.

So back to the question of why should we know these two objects? Like the previous lesson I said, if you are familiar with String you should know that *String is immutable*, in English is called *Immutable*. This means that when you create a Chain, the String is fixed and cannot be changed. But wait !!! In the

previous lesson, we had practice for Chain concatenation, Chain cut, upper / lower case conversion for String ... isn't that about changing String ??? Too confused. In fact, if you force to make a Chain of Changes, the system will create a new Chain for you, the old Chain (before being changed by you) will remain in the system and will be more at risk of becoming garbage., affecting the performance of the application.

Then *StringBuffer* and *StringBuilder* were born to meet your needs in cases where you *want to use Mutable String* . How did they do that? Please see the next section will clear.

II. What is the difference between StringBuffer and StringBuilder?

Another question that makes me even more considerate when it comes to these two subjects in this's lesson. Because the answer to this question is: *StringBuffer* and *StringBuilder* have exactly the same functionality and usage, but there is a slight difference in structure, that *StringBuffer is constructed to apply to behavior. multithreading (multithreading) help avoid disputes between the Flow (Thread)* , while *StringBuilder is made to applications in a stream only* . The concept of Thread or Multiple Thread will be discussed in the following lessons, so once again I just want to tell you, but to understand better, you must continue to follow the long series of lessons. by TechJA.

There are some speed comparisons on the internet that show that using *StringBuilder* is the fastest, then *StringBuffer* and then finally *String*. I believe this is true, but I haven't had the chance to verify it yet.

If you are still not sure about the differences between *StringBuffer* and *StringBuilder*, and still want to apply them to your current project instead of waiting to learn Thread, then you can use one of the two. *StringBuffer* for example.

III. How To Use StringBuffer And StringBuilder

Because I said that, *how to use StringBuffer and String Builder is completely the same* , so in this usage section, I just give the example of *StringBuffer*, you can completely apply the knowledge of This section takes the *StringBuffer* to the *StringBuilder* without any hassle.

1. Declaration and Initialization

Do you remember two ways to declare and initialize a String? If you forget, review the previous lesson to remember. Basically with String you have two ways of initializing, either "*primitive*" , or "*object*" type .

As for StringBuffer (and also StringBuilder) of this's lesson, it's a little different from regular String, that is only one way to initialize them, is to initialize type "*object*" . As mentioned in the String lesson, this lesson I would like to give the example of StringBuffer declaration without giving syntax.

```
StringBuffer strBuffer_1 = new StringBuffer(); // Creat a new StringBuffer //  
Create a new StringBufferm it can cotain 50 char StringBuffer strBuffer_2 = new  
StringBuffer(50); // Creat a new StringBuffer with initial content StringBuffer  
strBuffer_3 = new StringBuffer("Hello World!");
```

And since StringBuffer and StringBuilder are comfortable string-modifying constructs, you can rest assured that it won't affect your system's performance if you change a lot and constantly on the string. Here are the useful methods of StringBuffer and StringBuilder that you can take a quick look at.

2. Chain Concatenation - *append* ()

If with String, you can use function *concat* () or operator + to concatenate two strings together, but this way as we know it will create a new string that is a combination of two old strings. Then with this lesson the main chaining function is the *append* () function, which will have almost the same effect, that is to add a new string to the old string but the system still doesn't create another string. You see an example of how to use this function as follows.

```
StringBuffer str1 = new StringBuffer("Hello World!");  
str1.append(" Hello TechJA!");  
System.out.println(str1); // Result in console: "Hello World! Hello TechJA!"
```

3. Insert String - *insert* ()

This method inserts a string somewhere in the original string. The following example inserts the string "Java" after the string "Hello" in the above example, to form the string "Hello Java World! Hello TechJA! " .

```
StringBuffer str2 = new StringBuffer("Hello World!"); str2.append(" Hello
```

```
TechJA!"); str2.insert(5, " Java");

// Result in console: "Hello Java World! Hello TechJA!"

System.out.println(str2);

4. Replace - replace ()
```

If you remember, then in `String` we are familiar with the method named `replace()` already. But then the pass parameter of `replace()` is two strings, helping you to replace all the strings if existing in the original string with a new one. Through this's `replace()`, you must specify the start and end position of the original string to be replaced with the new string. As the following example (I got back from the practice request of replacing the string "`Hello`" to "`Hi`" in the previous lesson).

```
StringBuffer str3 = new StringBuffer("Hello World! Hello TechJA!");

str3.replace(0, 5, "Hi");

System.out.println(str3); // Result in console: "Hi World! Hello TechJA!"
```

Notice that the new string "`Hi`" only replaces the string "`Hello`" which has positions 0 to 5 in the original string. The string "`Hello`" after that is still in the original string, which is different from `replace()` in the *String lesson* which replaces all "`Hello`" to "`Hi`".

5. Delete String - `delete()`

This method, not present in *the previous lesson*, is used to delete the string from the starting position to the ending position in the original string. The following example removes the string "`Java`" from the original string.

```
StringBuffer str4 = new StringBuffer("Hello Java World! Hello TechJA!");

str4.delete(6, 11);
```

```
System.out.println(str4); // Result in console: "Hello World! Hello TechJA!"
```

6. Reverse String - `reverse()` An interesting method in this's lesson, is the chain island. See the reverse result from the example below.

```
StringBuffer str5 = new StringBuffer("Hello World! Hello TechJA!");

str5.reverse();

System.out.println(str5); // Kết quả in ra là "!skooB edoC wolleY olleH !dlroW olleH"
```

7. Checking the Buffer Capacity - `capacity()`

In the initialization step above you already know the ability to hold the number of characters StringBuffer and StringBuilder. If you create these objects empty, the initial default size is 16. See the following demonstration example.

```
StringBuffer str6 = new StringBuffer(); System.out.println(str6.capacity()); //  
Result in console: 16
```

This is the same when you *work with Arrays*, where you *allocate a memory* for an Array that can contain 16 elements, for example, without initializing the Array. Then with the above declaration of StringBuffer, too. Or with the above declaration having passed the initial number of characters that can contain.

```
StringBuffer str7 = new StringBuffer(30); System.out.println(str7.capacity()); //  
Result in console: 30
```

I continue to try adding the string and then recheck the *capacity()*.

```
StringBuffer str8 = new StringBuffer(); str8.append("Hello World!");  
System.out.println(str8.capacity()); // Result in console: 16
```

So when you declare and add a string, the initial default capacity is still based on the capacity to contain 16 characters, if the number of characters of the string exceeds 16, the capacity will automatically increase. This makes StringBuffer and StringBuilder quite flexible and performance of memory usage is also optimized.

In addition to the methods for StringBuffer and StringBuilder above, these are exceptionally flexible methods that the String in previous lesson did not have. These two new brothers of the Chain still have the same methods as the Chain side that you can bring to use, I just point out the face, that is.

- *subString(startIndex)*;
- *subString (startIndex, endIndex)*;
- *charAt (index)*;
- *indexOf (String)*;

We have just come to the knowledge that can be said to be the last of Basic Java. We will dive deep down into Java in the next lessons!

Lesson 15: An Overview of Object Oriented Programming

I feel like the last time has been a very long time when we step by step get acquainted with the Java language ... *no* . As you all know, right from the first Java lessons, I also said that this language was born for us to think in *Object Oriented* . Yet we have also learned fluently according to the old way of thinking, although sometimes we still interject some *object oriented* knowledge . This proves that *Object Orientation* is something very up close and easy to reach.

So what is *Object Orientation* ? How are the differences between *Old Direction* Thinking and *Object Oriented* Thinking, and are they complementary?

Why *Object Oriented* ? Clouds and clouds ... All these questions will be cleared up in the next series of TechJA. For this's lesson, since we have just set foot on the threshold of *Object Orientation* , it would be a good idea to go through the concepts and comparisons together before going into specific lessons.

I. Ways of Thinking in Programming

Object-Oriented or What-Oriented Programming is just a solution, a thinking in programming only . It is not sublime or complicated. It also doesn't depend on the Java programming language or anything. It is just a collection of methods and principles to help you think and organize the problem, thus making it easier to create a large and complex program.

And of course, with thinking in a certain *Programming Direction* , the programming language must also support the programmer with syntax according to the principles of that *Direction* . Therefore we have the programming language *Object-oriented* or *Nonobject-oriented* . As the Java language that you are learning is an *Object Oriented Language* .

Can say is that there are many documents describing how specific programming *Object-oriented* and how the programming *Non-object-oriented* . But with a few documents that I know, surely it is very difficult for you to understand this difference. So with my lessons, I invite you to approach the Programming *Directions* in a completely different way, that is, let's go through a simple example, and change our minds gradually to step by step explore how to build an "old" and *Object Oriented application* will be.

II. Building Applications In the "*Old*" Direction

Suppose it is required that you write a program that tells the user to enter the Radius of a Circle, and then it will print the Perimeter and Area console of that Circle. If you got to work with a quick code of the program, it should look like this.

```
// Ask user to give application radius of the circle  
System.out.println("Enter radius of the circle: ");  
Scanner scanner = new Scanner(System.in);  
float r = scanner.nextFloat();  
  
// Caculate perimeter of the circle here // ...  
// Caculate acreage of the circle here // ...  
  
// Print them info to console System.out.println("Perimeter of the circle is: ");  
System.out.println("Acreage of the circle is: ");
```

With the approach from *Lesson 1* up to now, it is very easy to get started in completing this program. To calculate the Perimeter and Area of a Circle, we just need to apply mathematical knowledge, you can see the complete code as follows.

```
// Create a constant PI final float PI = 3.14f;  
  
// Ask user to give application radius of the circle System.out.println("Enter  
radius of the circle: "); Scanner scanner = new Scanner(System.in); float r =  
scanner.nextFloat();  
  
// Caculate perimeter of the circle here float p = 2*PI*r;  
// Caculate acreage of the circle here float area = PI*r*r;  
  
// Print them info to console System.out.println("Perimeter of the circle is: " + p);  
System.out.println("Acreage of the circle is: " + area);
```

Oh, it's not difficult! So the way of thinking as we do with this example is called *Procedure Oriented Programming (POP)* , or it can be called *Structure Orientation* . Because we think in a way that breaks the request down into "procedures" , the procedures here are the possible actions of a program. For example, with the above example we have the procedure for *calculating Perimeter of Circle* , *calculating Area of Circle* . You can also separate each of these procedures into *Functions* (more on that later) to make the program easier

to see.

The story does not stop here. Suppose the requirement is raised. Requires the program to print out the Cylindrical Volume (this cylinder has the same Radius of the Circle as the Circle Radius above).

From this expansion request, we don't need to tell the user to enter any other radius, just enter the height. So with a *Functional-oriented* mindset , you will also cry easily, and... Boom! New code is born with some new procedures.

```
// Create a constant PI final float PI = 3.14f;  
  
// Ask user to give application radius of the circle System.out.println("Enter  
radius of the circle: "); Scanner scanner = new Scanner(System.in); float r =  
scanner.nextFloat();  
  
// Ask user enter height of cylinder System.out.println("Enter height of cylinder:  
"); float h = scanner.nextFloat();  
  
// Caculate perimeter of the circle here float p = 2*PI*r;  
// Caculate acreage of the circle here float area = PI*r*r;  
  
// Caculate volume of the cylinder here float vCylinder = dt*h;  
System.out.println("Perimeter of the circle: " + p); System.out.println("Acreage  
of the circle: " + arear); System.out.println("Volume of the cylinder: " +  
vCylinder);
```

Fortunately, the Perimeter, Area, and Volume functions in this example are very short. But you can see that if more and more requests arise, such as asking for the addition of Perimeter or Area or Volume for Squares, Parallelograms, Spheres, etc., your program procedures will expand. to the extent that it makes the program even harder to manage. Therefore, there is a need to have a new type of thinking that makes code management easier, and even shorten code time through re-use of computational procedures . So from that desire, *Object Oriented* thinking was born.

There are many documents that this *Functional-oriented* thinking method is *top-down thinking* , ie *top-down* style, or can also be understood as thinking in general to specific . . So what? That is, you will see the overall requirements of the previous program, for example, you immediately see the requirement to

calculate the Perimeter and Area of a circle, you will build the calculation procedures first. After approaching that whole, you find it necessary to write more smaller procedures that are easier to manage, for example you want to add the procedures for calculating the Square, the Decimal of a term. If it is possible to divide that small procedure into even smaller procedures, then divide it. That is top-down thinking.

III. Object Oriented Application Building

Object Oriented Programming (OOP). This new way of thinking is not directed at

procedures anymore, but towards objects . In fact, the objects will be entities or things that we can hold, can name. Generally there is state and behavior. Applied in software programs, the concept of object is still the same.

For example, with the above written program for calculating Perimeter and Area of a Circle. Then *the object that we need to care about is the Circle . So how to deal with the request? As you know, objects born in programming (or in real life) all have certain states and behaviors (actions), where the Circle is responsible for calculating the Perimeter and Area of the plate. my body . Ie procedures in thinking User procedures The above will be all assigned to the Process Circle. You can even construct the Circle itself asking the user to enter a Radius and save it. Then call Circle itself prints the results Perimeter and Area to the console as well. Strong!!! As I illustrate with the code below, you do not need to know exactly what this code is like, just know the idea of what the Object Orientation is like.*

```
// Creat an object Circle // and order them to task Circle circle = new Circle();  
  
// cirle itself ask user to enter radius circle.enterRadius();  
// Circle itself process caculating perimeter circle.caculateP()  
// Circle itself process caculating area circle.caculateA();  
  
// Circle itself print infomation to console circle.printP(); circle.printA();
```

Do you realize how neat and manageable the code is? I would love to see code like this.

The difference is huge when the procedures for *calculating Perimeter and Area* are large and complex. Which Entity's procedures are then assigned to that

Entity, that is, you build those specific procedures within Entities (you will learn how to build this later), and when needed Which procedure, we just call the procedure through the declared object, like the *Circle* object above.

For you to specify Circle must have its procedures. Then adding a cylinder is just the declaration of adding a new object, with corresponding new procedures. In addition, the Cylindrical object has the same properties and procedures as the Circle, so we can completely reuse the procedures declared on the Circle. The reuse of these procedures is called *Inheritance in Object Orientation*, through the designation of Cylindrical as a child of Circle (we will consider the *Inheritance* relationship. This is in the following lessons). The fact that a Cylindrical is a child of a Circle makes it possible to have functions whose father is a Circle defined, so managing and reusing code is very easy. Later, if there appear more Sphere, Image abc, or Figure xyz, then we can completely create objects and they will have some relationship with each other, with specific or successive procedures. surplus of each other.

I also have a code example for the desire to calculate the Cylindrical Volume as follows.

```
// Creat 2 object Circle và Cylinder // and order them to do their own task Circle  
circle = new Circle(); HinhTru cylinder = new HinhTru();  
  
// Circle and Cylinder themselves ask user to enter infomation  
circle.enterRadius(); cylinder.enterRadius(circle.r); cylinder.enterHeight();  
  
// Circle itself calculate Perimeter circle.calculatePerimeter(); // Circle itself  
calculate Area circle.calculateArea();  
  
// Cylinder itself calculate Thẽ tich cylinder.calculateVolume();  
  
// Circle và Cylinder itself print info to console circle.printPerimeter();  
circle.printArea(); cylinder.printVolume();
```

If the above *Procedural Direction* is above-bottom thinking style. Then this *Object*

Orientation, with the first approach to each Entity such as the Circle, the Cylinder as in the above example, then we build the procedures for it, add relationships, edit the roles. for each object when system requirements are

increasingly complex. This way of thinking is called *bottom-up thinking*, that is, from the bottom up, or from concrete to general.

That's all! Core *Object-Oriented* Thinking is all. You saw like programming *User object* then is not it. Don't like it, too, because without *Object Oriented*, you won't be able to build big applications in the future. The above is just a brief example of *Object Orientation*, everything that mustache and its strengths we will gradually talk about in later lessons.

IV. So in short, What Is Object-Oriented?

As you have been familiar with above, this section is closed. That is, has been called *Object Oriented*, which means wanting you to always think things according to the object.

For example, with the requirement of building *Student management software*, Schools, Classes, Students, Teachers, even Tables, Chairs, Projectors, Subjects, ... can all become objects for us to operate. use in management. As long as the object is present within our application.

Or with the requirement of building a *Tour management*, Notes, Themes, Accounts, Itineraries, ... are all objects that you can organize.

This's lesson just stops at a review of the ways of thinking, to help you have in mind an idea of the *Directions* in programming. Then the following lessons, we step by step concretize these concepts into a complete knowledge. One last thing I need to note is, *Object Oriented* programming is not a completely new type of programming to replace the old *procedural direction*. That *User object* launched to bring you a new tool to build applications faster. You must use *Object Oriented* to build your product, but you don't have to give up a *Procedural* mindset. Inside the *Object Orientation* there is a *Procedure Direction* They all contribute to a complete program that is tightly structured, easy to read, easy to maintain, and easy to upgrade.

Lesson 16: Objects & Classes

With the previous lesson, we became familiar with the basics of Object Oriented Programming. At that time, we have approached to change the way of thinking

from the old Functional Orientation to this new Object Orientation . And you have certainly grasped the core of this Object Oriented thinking, which is to always focus on Objects .

So this we will go into that focus together, The Object. And speaking of *Objects* , we also have to say *Class* .

I. Concept of Objects (Object)

As you also learned in the introductory lesson the other day, *Objects* are entities or things that we can hold, or can name. It sounds rambling, but it's actually... rambling. I'm just joking, actually because Object Orientation is also an open way of thinking. That is depending on your conception to manage only which Audience.

For example, with car business management software. Then you choose Car as an object to manage. But I say that the Customer is also an object. And Invoice is also an object. See, all that appears in an application, that you want them to be an entity to manage, you can just think of it as an object.

Also, if you have selected an entity as an object, then that object must have *States* and *Acts* .

1. Subject's Status

The state of the object denotes the *characteristics* , or *characteristics* of the object.

For example, when you "grab" a Car object in the car management application above to see it, you will see the *status* of this object such as: *this car is blue, belongs to Ford, 7 seats* . But if you "hold" another car, you will see other *states* , such as: *this car is white, belongs to Toyota, 4 seats* .

You see, the thinking in Object Oriented Programming looks very realistic, doesn't it. Remember these *states* of the object, we will need it in the examples in the next knowledge below.

2. Subject's Behavior

Other than *the state* are the characteristics of the object. The *behavior* is the *action* of that object, or can understand that is the *operation* that the object has a

responsibility to perform.

As you are familiar with the opening example in the previous lesson, as we gradually pass the computation code in *main ()* function into the Circle object. Then the computational functions become the *behaviors* of this Circle object. This means that the Circle is responsible for doing the Perimeter and Area calculations. And of course Circle is not responsible for the Perimeter and Area calculation actions, it will belong to the control of Square object !!! You have a better understanding of the behavior of each object, right?

II. Class Concept

If you are familiar with the concept of *Objects* . Then the concept of *Class* will be a little simpler. *Classes are seen as a template for creating Entities* . It's short like that, but it's short then don't understand anything. Actually, it's not that difficult, you must distinguish between Class and Object. Let's go slowly through the following examples.

1. Example 1: Baking

Let's forget about programming for a bit, let's get started on making butter cookies as follows. Let's say you already know all the baking recipes. And I asked you to help me make them with this heart shape so I can take away "*that person*" .



But you say that knowing the recipe is not enough. If you want you to make exactly the same shaped cookies, *give you the mold* . Well, this is *the mold* , I have it prepared, if you want.

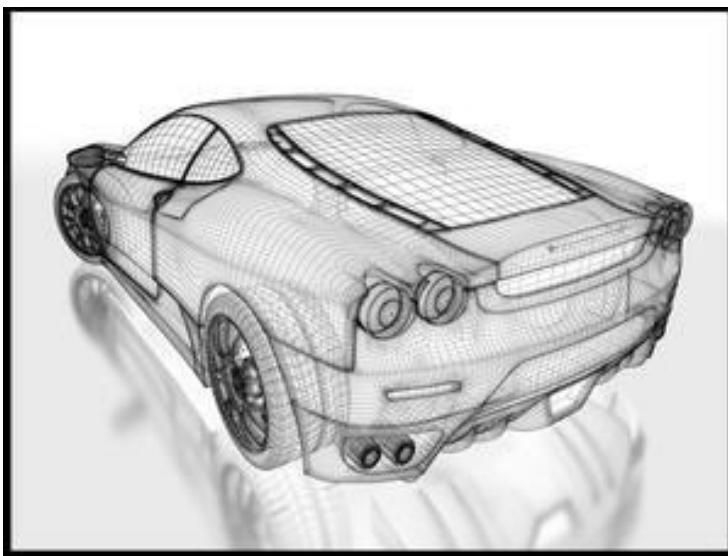


2. Example 2: Designing Car Models

I want you to ramble a bit through the next example. Suppose you are a 3D artist, specializing in painting finishing textures for cars. With the request that you produce the finished car with the following colors.



Oh of course you can. But first you ask for a *concept drawing* of this car, then you can wear them in colors. Yes I have also prepared for you, here is the *concept*.



What is the message of the examples above. I want you to imagine from the fact

that the Objects that we need to pay attention to from these examples are tangible things that can be seen or interacted with by the user. As each finished *biscuits butter* in example 1 is an *object* separately. Or, each finished *car with a different color* in example 2 is also a separate *Object* .

Then the *stereotypes* that make up those products, like the *heart shape* , or the *concept of the car* , are the *Class* that we need to learn.

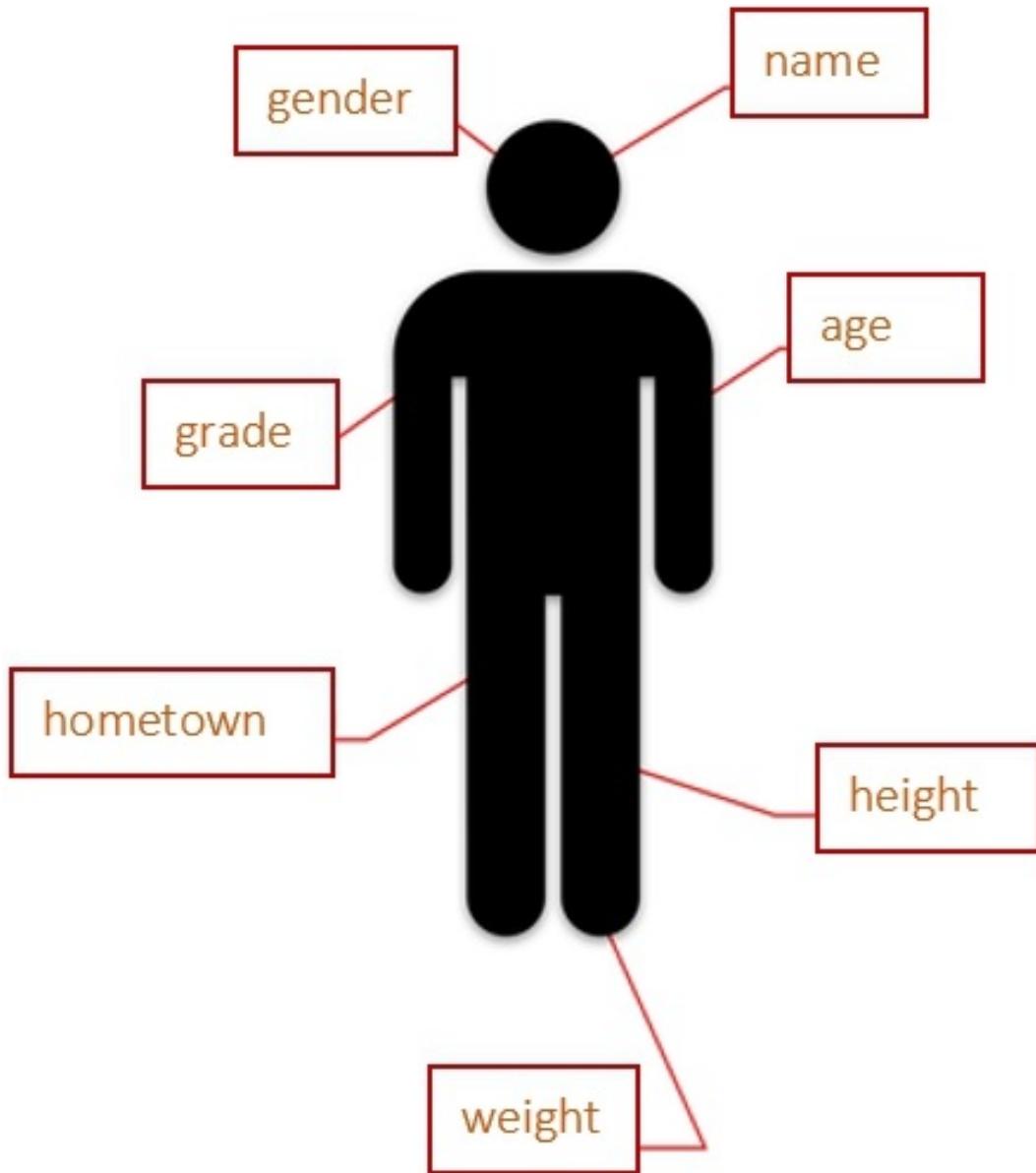
3. Example 3: Doing Student Management Software

Now, no more rambling examples. We return to the realm of programming with the Student Management example. Suppose in the school that we need to manage there are the following Students.



Apply from examples 1 and 2. Students are definitely the subjects that need management. So what will be the *template* for these Student objects. In this case, what's a little abstract Mold tiny, it will be a generic image with labels

like: *Gender* , *Name* , *Grade* , *Age* , *Hometown* ,



Height ,

Weight . As shown below.

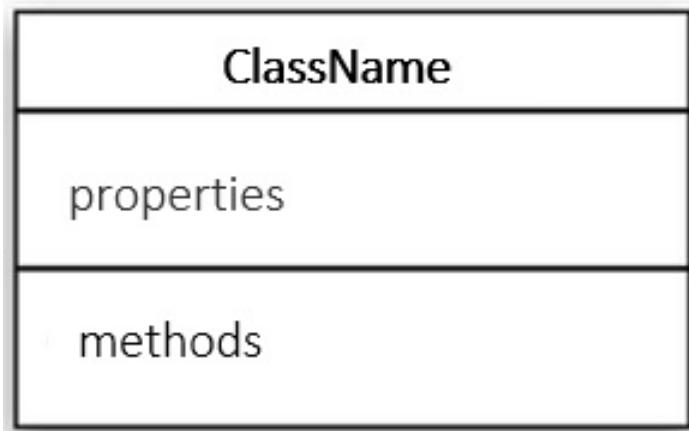
From that *stereotype* , or we already know that it is *Class* , it helps to create many different Student Objects. So you have understood the concept of *Classes as a template for creating Entities* , right? And if the *Object has States and Behaviors* , the *Class also has properties and methods* respectively.

From the previous lesson up until now, we just rambled to talk and talk. Now it's time to code. But before we get into how to code a Class, let's go through this section first.

III. Class Declaration

1. Class Visualization

When you want to bring a certain entity to an object to manage, you have to template that object. That is Class. And before you start creating a Class, you must visualize that Class with three main components as shown below.



2. Class Declaration Syntax

Once you have envisioned a Class with the three components above, the representation in Java code will follow the following syntax.

```
class_class {  
    properties;  
    methods;  
}
```

Inside:

- *class* is just a keyword that indicates you are declaring a Class.
- *class_name* is an identifier for that Class, the naming rule for the Class is the same as that for the *Variable*. But please note a little that this *class_name* should capitalize every first letter of each word, for example, Circle, SinhVien, HoaDon, ... to distinguish from the name when you declare an object will be said in the section. down here.
- *properties* and *methods* will be discussed in more detail in the following lesson.

- *Practice # 1*

In this exercise we will proceed to build the Circle class that in the previous lesson I have an example for you to see.

Note that this exercise does not require you to understand all the lines of code, you just need to look at the general structure of the Circle class as what I mentioned in this's lesson. All the expensive details of the Class will be discussed in the following lessons.

We temporarily enter the code in the same place where the *main ()* function is stored , and below this *main ()* function is as follows.

```
public static void main(String[] args) {  
    // ... you don't have to care what is written here }  
    // Create class Circle class Circle {  
  
        // properties of Circle final float PI = 3.14f; float r;  
        float p;  
        float area;  
  
        // methods of class Circle  
        void enterRadius() {  
            System.out.println("Enter radius of Circle: ");  
            Scanner scanner = new Scanner(System.in);  
            r = scanner.nextFloat();  
        }  
  
        void calculateP() { p = 2 * PI * r;  
    }  
  
        void calculateArea() { area = PI * r * r;  
    }  
  
        void printP() { System.out.println("Perimeter Circle: " + p); }  
        void printArea() {  
            System.out.println("Area Circle: " + area);  
        }  
    }
```

I say a little more. With the declaration of a Class as above:

- *class_name* is now *Circle* .
- *main_properties* are variables and constants, including *PI* , *r* , *p* , *area* .

- the main *methods* are the *input* functions *BanKinh ()* , *calculateP ()* , *calculateArea ()* , *printP ()* , *printArea ()* .

IV. Declaration And Object Creation

The declaration of an object does not have a general syntax, it depends on the specificity of each Class of that object, which we will talk about in the following lesson. But remember that one thing in common is that the *new* keyword is required in the declaration and initialization of these objects.

Let's practice declaring the Circle object, this is the code of the previous lesson. •
Practice # 2

Try declaring the Circle object inside the *main ()* function as follows.

```
public static void main(String[] args) {  
    // creat object Circle and order them to do their task  
    Circle circle = new MyFirstClass(). new Circle();  
  
    // circle itself ask user to enter its radius circle.enterRadius();  
  
    // Circle itself calculate Perimeter circle.calculateP();  
    // Circle itself calculate Area circle.calculateArea();  
  
    // Circle itself print result to console  
    circle.inP();  
    circle.inArea();  
}
```

Again don't focus too much on how to use functions in an Object, you will get used to Objects soon.

You just need to pay attention to a few places.

- The line of code *Circle circle = new ...* is where you declare an object is *circle* . This *circle* object is created from the class *Circle* . As you know how many other Round objects you can create from this *Circle* class (like the code example right below). You can even create the Array of Circular Objects, which we will look at in another lesson.

```
Circle circle1 = new Circle();  
Circle circle2 = new Circle();  
Circle circle3 = new Circle();  
Circle circle4 = new Circle();
```

- The *new* keyword as I said is required when declaring and initializing these object data types, it is different from when you declare a variable of the primitive data type that I mentioned in *the lesson about variables*. After what the *new* keyword is, I will say it clearly in the next lesson.
- The remaining lines of code are calls to the *properties* or *methods* of that object. Now, if you execute this line of code, you should be able to interact with the console.

We can end this's lesson here. So you have just become familiar with the two most basic concepts in Object Oriented programming, which are *Class* and *Object*. And together define and instantiate the object *circle* from the class *Circle* for practice. We will talk more deeply about the structure and how to use Objects in the next lessons

Lesson 17: Constructor

What a beautiful day this was to take a look at Java's constructor concept and usage.

Of course you will find that this lesson is talking about some kind of method, so why don't I incorporate it into the lesson of *methods*. Well, not just you, many of you have questioned me like that. But you know, the method that you are familiar with this will be a bit more special than the methods that we have talked about, especially, please see the content below. And because it's special, it's a bit different, it's important, so I separate this type of method into a separate lesson, so that you have a more comfortable, independent, ambiguous approach between normal method and this constructor.

I. Constructor concept

The constructor, or call the *constructor*, is also okay, you can also call *Constructor*, I will use the constructor for short.

In essence, this constructor is also a method, but it is special that, as soon as you initialize an object with the keyword *new*, the constructor of that object will be immediately called automatically. This means that if with a normal method, you have to call it through the dot operator (".") Then that method will be executed, but with the constructor, as soon as the *new* keyword is compiled, the

system will execute a corresponding constructor of the object, depending on which constructor you specify.

The main purpose that the constructor offers is nothing but the effect of *Initialization*. Constructor helps the newly created object have the opportunity to initialize values for properties within it. Or you can help the object to call other corresponding methods to initialize the logic inside the object.

Before understanding how to use a constructor, let's look at how to declare them.

II. Constructor Declaration

First of all, I would like to talk through the syntax for a constructor, so that you can compare it with the declaration of a normal method in *lesson 18*, see if there is any difference. The syntax of a constructor is as follows.

```
[accessibility] method name () {
```

```
// Lines of code }
```

So you can also see the difference, but I also go through the components inside the above syntax for it to be clear.

- First, the *constructor has no return type like normal method*.
- *Accessibility* - We will talk about this in a different lesson, along with *access* to the normal properties and methods of a class. However, in this's lesson, I will use *public* for constructors, it means that these constructors can be used anywhere.
- *method_name* - Unlike the normal mode, *the name of the constructor must along with the class name*. To help distinguish between what is a constructor and what is a normal method.
- *parameters* - This is the same as normal method, nothing more to say.

1. Creating Constructors For The Circle Class

We still take the *PackageLearning* project from *the previous lesson*, and put together constructors for the objects.

Take out the *Circle* layer as a white mouse. You pay attention to the two constructors that I give in the following example. I just declared it, there is no code inside these constructors, the rules of declaring these constructors

completely follow the bullet points above.

Try to look at these two constructors and remember and re-code yourself, remember to re-code, don't copy / paste, you will learn a lot through the lines of code for this constructor, believe me.

```
public class Circle {  
  
    / *** * Demo how to declare constructors, * properties and methods remain the  
    same as in lesson 19, * Note the commented constructors * /  
  
    final float PI = 3.14f;  
  
    float r; float p; float area;  
  
    // A constructor, note there is no return type, // and this constructor has no  
    parameters passed public Circle () {  
  
        // We initialize something later }  
  
    // Another constructor, also has no return type, // but has one parameter passed  
    public Circle (float r) {  
  
        // We initialize something later }  
    public void enterRadius () {  
  
        System.out.println ("Please enter the Radius of a Circle:"); Scanner scanner =  
        new Scanner (System.in); r = scanner.nextFloat ();  
  
    }  
  
    public void calculatePerimeter () { p = 2 * PI * r;  
    }  
  
    public void calculateArea () { area = PI * r * r;  
    }  
  
    public void printPerimeter () { System.out.println ("Circumference of a Circle:"  
        + p); }  
    public void printArea () { System.out.println ("Area of a Circle:" + area); } }
```

Through the example above, I have a few more notes to supplement the other four bullet points. I want you to finish the code through the constructor before talking about this to avoid confusion.

- *In a class, you absolutely can have multiple constructors, each constructor so must other parameter passed* (and not another name then, as on the other have to say that the constructor must have the same name as the class, so the constructor all must have the same name). You can review the above example to see that there are two constructors, but you can create many more constructors as well.
- *With a class has multiple constructors, you absolutely can from this constructor call to the constructor*, the call does not create a new instance of the class, but the main purpose of this is to take advantage of the code line initialization of the constructors only. This issue you will understand more in the lesson on how to use this keyword in the following lesson,
- No matter how many constructors that class has, *when declaring the object, you must specify one and only one constructor*. This is different from normal methods, as you can call as many methods as you like. In the following step, we will see how to specify a constructor for the object.
- *A constructor is executed only once when the new keyword is called. You cannot execute a constructor for the rest of the object's life*. If you want to implement a constructor again, then you must use the *new* keyword, then you have created a new object. This is also different from other normal methods that have callback capability. Therefore, if you need to initialize values, just initialize them in a constructor.
- And one more point is also quite important. What if you forget to declare the constructor for a class? Just like in previous lessons so far, you only create properties and methods for the class, not constructors for them! Then, *the system will always implicitly create a constructor with no parameters passed, nothing inside that constructor*, just like the first constructor of the Circle class in the above example. Thus by default we will always get a constructor from the system.

In the next exercise, you will add code into constructors.

2. Practice Initializing Values Through Constructor

Now that you make sure that the *Circle* class is open, try to code the following

lines into two constructors.

```
public class Circle {  
  
    / *** * Demo how to declare constructors,  
    * properties and methods remain the same as in lesson 19, * Note the  
    commented constructors  
    * /  
  
    final float PI = 3.14f;  
  
    float r;  
    float p;  
    float area;  
  
    // Constructor has no parameters passed public Circle () {  
    nhapRadius (); // Try calling nhapRadius () }  
    // Constructor has one parameter r passed public Circle (float r) {  
    this.r = r; // Assign variable r to attribute r }  
  
    public void enterRadius () {  
        System.out.println ("Please enter the Radius of a Circle:"); Scanner scanner =  
        new Scanner (System.in);  
        r = scanner.nextFloat ();  
  
    }  
  
    public void calculatePerimeter () { p = 2 * PI * r;  
    }  
    public void calculateArea () { area = PI * r * r;  
    }  
  
    public void printPerimeter () {  
        System.out.println ("Perimeter of this circle is: " + p); }  
    public void inArea () {  
        System.out.println ("Area of this circle is: " + area);  
    }  
}
```

You can see that.

In the first constructor there are no parameters passed, in which I call the function that calls for the radius. Remember this content of the first constructor, in order to execute the program a little while, you will understand why.

In the second constructor there is a parameter passed as variable *r* , when we receive this variable, we assign it to property *r as well* .

If you still do not know what the constructor does, do not be discouraged, continue reading the next section, the implementation of the constructor, you will understand more and more.

III. Declaring Objects Through Constructor

Remember, how did you declare the object *circle* and *rect* in the previous lessons? Is that so?

`Circle circle = new Circle();`

`Rectangle rect = new Rectangle();`

As I said, if you do not declare any constructors for the *Circle* and *Rectangle* classes , the system actually has declared you a constructor for each of those classes as follows. `public Circle() { }`

`public Rectangle() { }`

And so, when you declared these two objects, you called them through the default constructors, *new Circle ()* , and *new Rectangle ()* . You see no connection.

So back to the *Circle* class that you worked on earlier, we have declared two constructors inside the class. And I also said that you can only execute one and only one constructor, so you will have one of two ways to initialize *Circle* as follows.

```
// Create object circle1 by using first Constructor
```

```
Circle circle1 = new Circle();
```

```
// Create object circle2 by using second Constructor Circle circle2 = new  
Circle(10);
```

With these two constructors, *how will circle1 and circle2 differ*, let's go through the exercise.

- *Practice Declaring Circle Through Constructors*

Let's go back to the *MainClass* class to initialize *Circle* through constructors.

Let's code together as follows.

```
public class MainClass {  
    public static void main(String[] args) { // Create object circle1 by using first  
        Constructor Circle circle1 = new Circle();  
        // Create object circle2 by using second Constructor Circle circle2 = new  
        Circle(10);  
  
        // Calculate and print result for circle1 System.out.println("===== circle1  
        ====="); circle1.calculatePerimeter(); circle1.calculateArea();  
        circle1.printPerimeter(); circle1.printArea();  
  
        // Calculate and print result for circle2 System.out.println("===== circle2  
        ====="); circle2.calculatePerimeter(); circle2.calculateArea();  
        circle2.printPerimeter(); circle2.printArea();  
    }  
}
```

This means that we create two objects *circle1* and *circle2* from class *Circle* . In the *Circle* class we have two constructors, *circle1* is created through a constructor with no parameter passed, *circle2* is created through constructor with the parameter passed as a *float* variable .

Now if you run this program, you will see that once the console asks you to enter the radius of a circle. That's because the *circle1* constructor calls the *nhapRadius()* method . As soon as *circle1* is declared through *Circle circle1 = new Circle();* The corresponding constructor (no-parameter constructor) is immediately implemented.

Similarly, for *circle2* is declared by *Circle circle2 = new Circle(10);* , this means that this object has selected a constructor that has a *float* parameter passed to the initialization, and since you pass the value *10* immediately on that constructor, inside the constructor body, it assigns this value to the property *r* , and then when you call *calculatePerimeter()* and *calculateArea()* on this object *circle2* , then the value *10* will be used.

The result is printed to the console as shown below, when you only enter 5 for the radius *circle1* , *circle2* has been initialized with radius of 10 already.

Please enter the Radius of a Circle:

5

===== circle1 =====

Perimeter of this circle is: 31.400002

Area of this circle is: 78.5

===== circle2 =====

Perimeter of this circle is: 62.800003

Area of this circle is: 314.0

You already understand the constructor, right?

Above is the knowledge of constructor. You should understand that a constructor is also a method, but it has some differences as I have outlined above.

Constructors are very commonly used, to give an initialization value to an object immediately when that object is initialized.

Maybe the above ideas are not completely complete with a constructor. Or the way I present is somewhat confusing. But don't give up, keep learning and the result will come!

Lesson 18: Getting Started With Inheritance

So we have passed through many important knowledge in object oriented programming, such as *properties* , *methods* , *constructors* . But there is one kind of knowledge that can be said to be the quintessence of object oriented, which we will approach starting in this's lesson that will give you a way to use and organize classes in your application in such a way. The knowledge is completely more advanced and effective than the ways that you have been familiar with in previous lessons, which is the knowledge of Inheritance.

If you are new to Java and object oriented, inheritance will make it a little more difficult. At this point, Java code is no longer as important as how you organize the structure for classes or objects within your application. In my experience, after getting acquainted to the inheritance, there will be countless questions to bring to you. Yes, you will wonder when to inherit. There are also friends who understand inheritance, but the organization of inheritance wwidthly "*relatives*" of them, making the inheritance become more confusing. Then when should we block the inheritance of any given class? etc. These questions I will try to cover thoroughly in this series of lessons on inheritance, so that you have a best way to apply inheritance to your product.

I. Familiar With Inheritance

1. What Is Inheritance?

Inheritance in object-oriented programming refers to a relationship between objects, some say this relationship is *Father-Con*, some say it is an *Extended* relationship. People seem to like the concept of *Father-Son* more, but I see *Openness*, and most closely, it is better to understand that *Inheritance* is always better. Because it is essentially a reuse, in some cases it extends the properties of an object from some other object.

So, to understand a practical way, suppose we first have a certain class, this class can be written by us, or "gleaned" somewhere, we temporarily name this available class. is *A*. Then, in order to take advantage of *A*'s functions or properties without having to rewrite (or copy over, maybe piracy), we build a new class that inherits from *A*, we call the class. this new *B*. Then our *B* will have available methods and properties that *A* has. There are times when it is not for the purpose of *reusing A's values*, but because some of *A*'s values do not match *B*'s needs., That is the inheritance from *A* also helps *B* have the opportunity to complete the (also known as extensions) values were not right of *A* that does not alter the nature of the *A*. This, we are focusing on *Reuse*, *Expansion* purposes, we will talk about in the next lesson.

2. Why Must Inherit?

Through my above ideas, perhaps you also understand the reasons why they must inherit right.

Well, *the main purpose of inheritance is to reuse, and to further extend, properties and methods available from an object*.

That's all, the purpose of inheritance is not sublime at all, but the effect it brings is very great. By reusing these existing ones, your project structure will look more professional and your code will be easier to read. In addition, inheritance also helps you reduce the burden of coding a lot, because you take advantage of the existing code of other classes.

II. Inheritance In Java

We are talking about concepts in general. So how do we demonstrate inheritance in Java? In Java, to express a class that you want to inherit from a certain class,

you use the *extends* keyword .

Take a look at the example below, you should not rush to code, later in the practice we will code together.

```
class Circle {  
    float r;  
    float getRadius() {  
  
        return r;  
    }  
}  
  
class Cylinder extends Circle {  
}
```

The above code is an explicit representation of inheritance, where the *Cylinder* class inherits *Circle* with the keyword *extends* . And according to the law of inheritance, *Cylinder* can inherit the values (properties and methods) that *Circle* has declared. Through such inheritance relationship, one can call the *Circle* class *the Base Class* , or the *Parent Class* . And *Cylinder* class is called *Derived Class (Derived Class)* or class *Child (Subclass, Child Class)* .

Usually, the superclass, or base class, is the class that holds the common values, or the most base values, for the subclasses. So if you have multiple layers with certain similarities, like the Circle and the Cylinder in the example above, these layers all have round faces (cylinders have two round faces), so you can use Circle as the layer. base (since it contains the minimum values that the Cylindrical can take advantage of, in this case the round face itself). Or there are cases where there are many classes with similar values, in which you can combine into a single base class and then the subclasses just inherit and reuse those similar values without Need to declare something more, as in the exercise below, I will create a class *Geometry* is the most basic for the classes *Circle* , *Square* , *Rectangle* , ...

Above is an example of when you need to inherit, but now I would like to list some important points in the process of organizing your inheritance.

- A class is only allowed to inherit from one and only one parent class .

- Although not inherited from many superclasses, but the parent class that this inheriting object can inherit from another superclass, you can call this other

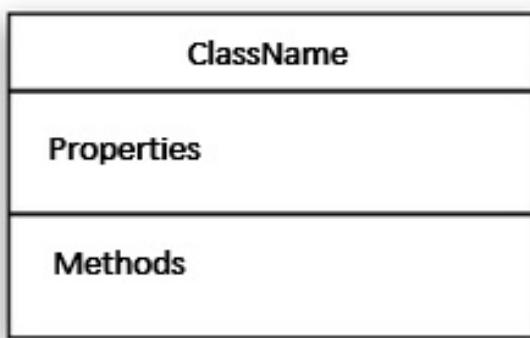
superclass as "*grandfather class*" for easy remembering, and surely there could be a "*great-grandfather class*" if the "*grandfather class*" inherits another class.

- If a class does not declare inheritance at all (like the classes we have practiced in the previous lessons), then the system will default to see it inheriting from the *Object* class . In the next lesson, we will talk about this *Object* class .

III. Familiarize With Class Diagrams

The concept of inheritance in this's lesson is just that. But before going into specific practice, I invite you to familiarize yourself with a divine diagram, which if you are a mainstream programmer, you cannot ignore. The main thing about this section is to talk about how you see and understand the diagrams. Because our project becomes more and more complex, there will be many and many classes, they use each other, inherit each other. And so without a diagram, we would not be able to describe all these relationships in words.

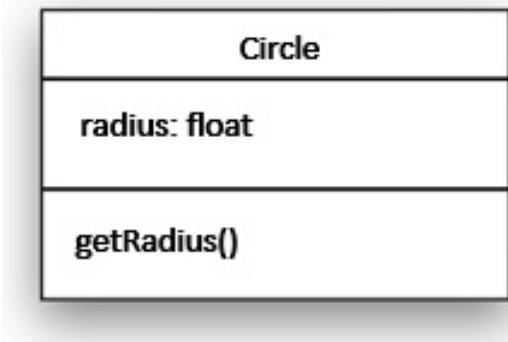
The class diagram of this section was taken from the *UML* 's *Class Diagram* construction principle . UML is a set of many different principles for software system specification and design. Say that so that you understand this is a standard principles compliance diagram, if you understand and follow these principles, you will create a common speaking model, which everyone who reads and understands you want Shows what to your software. You know, you are a little bit familiar with this diagram, because I used it in *lesson 16* , then I want to express a class with three main components as follows, and the shapes you see is exactly how the class diagram represents a class instance.



The shape and color of the blocks in the class diagram may differ in our lesson and in other documents, depending on the tool for drawing it. But even though

they differ in appearance, in general, the data that each diagram shows must comply with a cube principle with the above three components.

I take the example of the *Circle* class like the code above, then when represented as an entity in the class diagram, it looks like this.

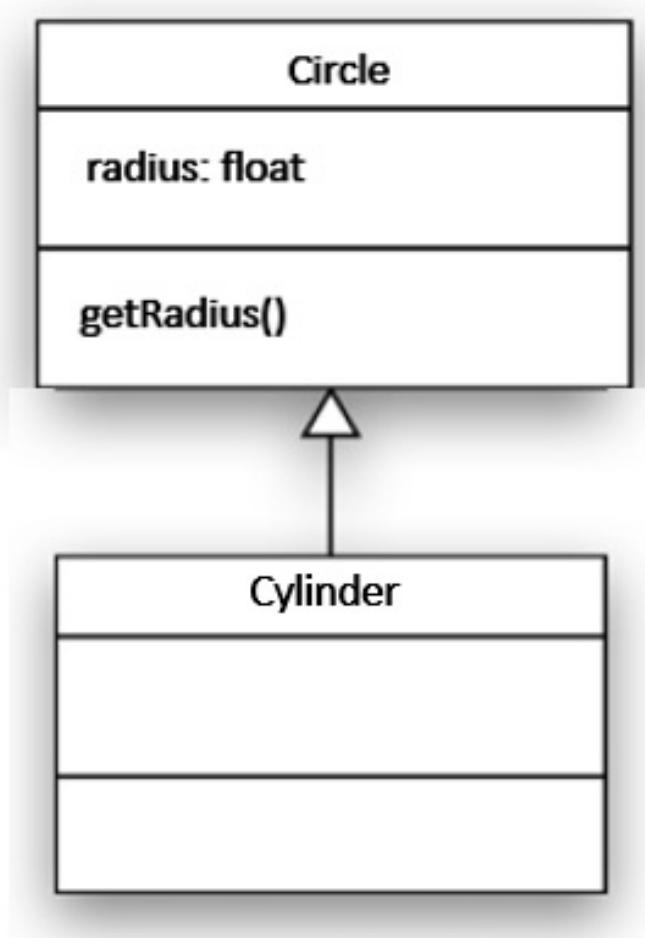


Looking at the diagram, you know right away that you need to build a class named *Circle*, this class has a property named *r* with data type *float*, and a *getRadius()* method returns *float*. Although it has a static value, it does not show the content or relationship of the elements inside a class, but

surely you can easily understand and visualize how to build a class based on a diagram. This is how.

Next, I continue to talk about how this class diagram shows inheritance. With the expectation that the *Cylinder* class would inherit from *Circle*, we can show through this relationship class diagram with an arrow mark as follows. You must pay attention to the empty arrow mark as shown, if you use a different type of arrow, it will cause confusion with other relationships.

One more idea of the diagram, for example in the case in *lesson 18*, the *Circle* class uses the *Point* class to make a property, which is named *prescriptionDo*, so the diagram will show this usage as follows: .



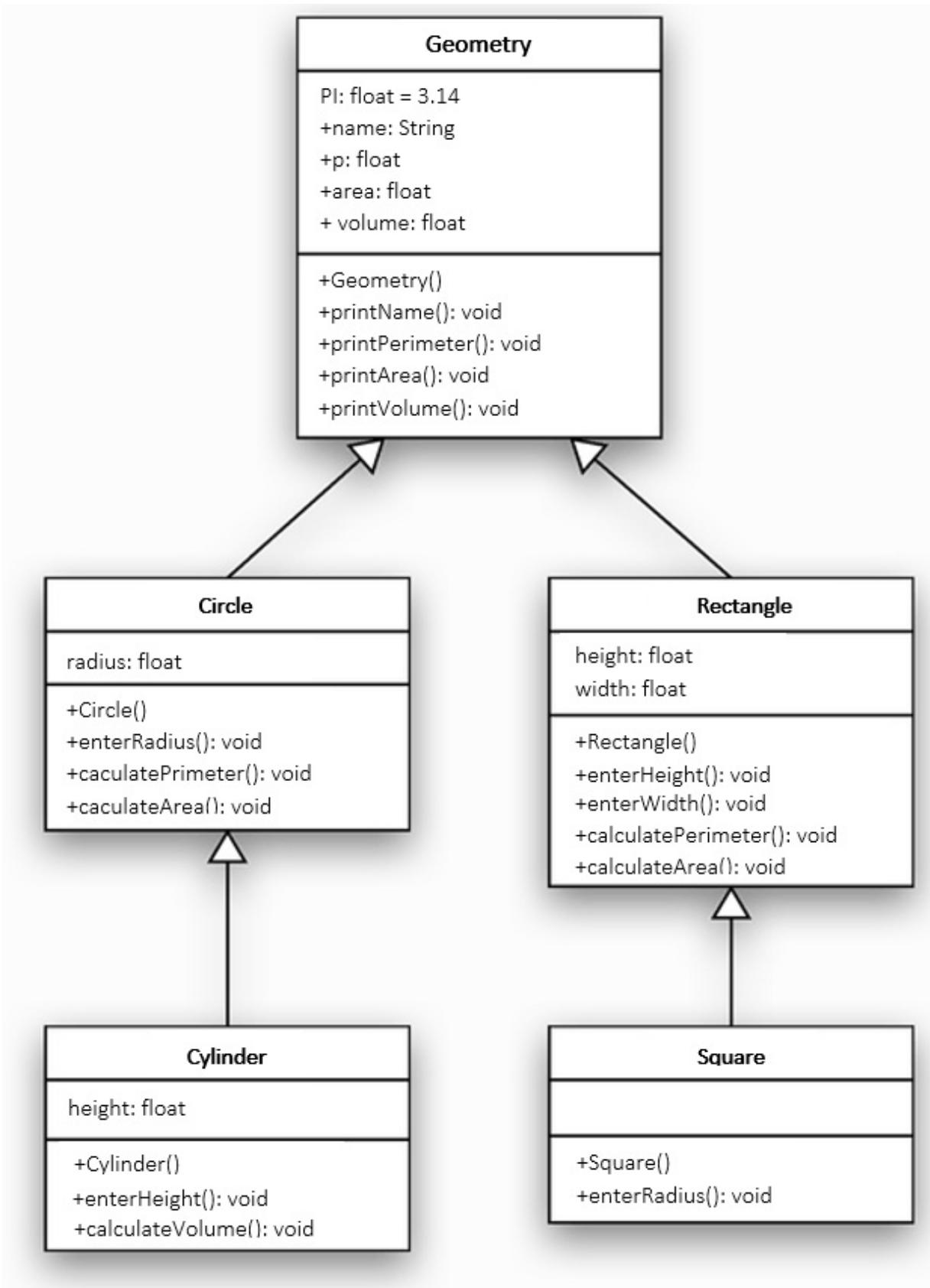
Everything is really clear, isn't it? Done, with this's lesson, I only briefly presented about inheritance and class diagram like that. We will add knowledge and symbols to this diagram in the next lessons. Now we need to practice getting used to.

- *Inheritance Practice*

We will continue to build applications to calculate geometrical values for *Circle* , *Cylinder* , *Rectangle* , and *Square* . You should also know that, if you do not apply the inheritance knowledge of this's lesson, you will still build the results of this exercise perfectly with the knowledge of OOP in the previous lessons. can try. But with the application of inheritance, as I said, you will save significant lines of code. Then invite you to try building a new project with me.

First of all, I invite you to look at the class diagram of this's lesson (with

inheritance applied) as follows, please consider a little bit (note the + signs in front of each method or property in the diagram are declared with the keyword *public* , this is *accessible* to the class values, which we will talk about later).



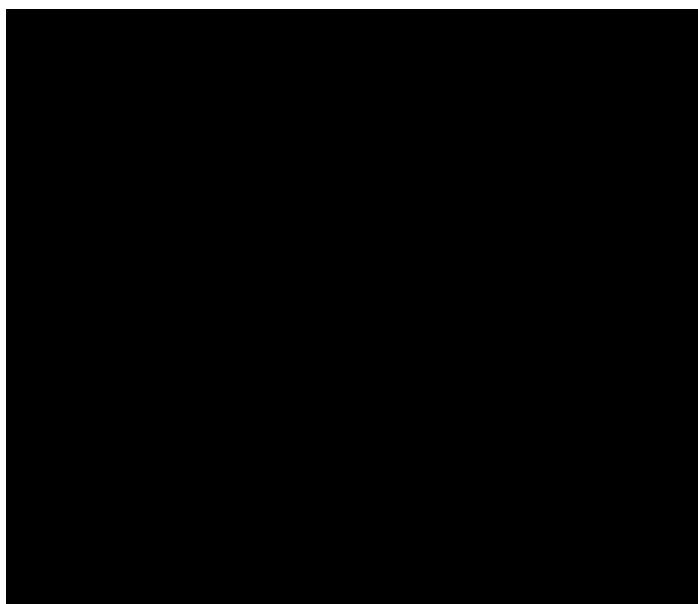
Before going into the official, I will look at the diagram and detail each class, then show you the results of running the program, and finally the code of each class. Please take a look at the code of the classes quickly, but rely on the diagrams and results and try to code, each person will have a different way of code, you do not have to code the same as me, as long as your program running fine is fine.

- Grade *Geometry* . This is the superclass of the remaining classes, or means the most basic class. Since it is a base class, it would be best if it contained properties or methods that would be useful to subclasses, or it could be said that the subclass could effectively inherit those values. For example, the *PI* constant I will declare in this class. The *name* attribute is shared, but will be specifically defined by subclasses according to their name. The *p* , *area* , *v* are the same, although they are defined in general, the subclasses will contain different values. The methods of this superclass are also meant to be used by subclasses, so they will have a specific function body, like *printName ()*. will output *name* out console. Or *printPerimeter ()* , *printArea ()* , *printVolume ()* will also output the variables *p* , *area* , *v* respectively.
- *Circle* class . A subclass of *Geometry* . As you know *Circle* will inherit the values from its superclass. In addition to the properties it gets from the superclass *name* , *p* , *area* , *v* , it also defines its own special attribute *radius* . You can initialize the variable *name* for *Circle* in the *Circle ()* constructor . The methods *enterRadius ()* , *calculatePerimeter ()* , *calculateArea ()* are not declared in the parent class, *Circle* designs itself.
- *Cylinder* class . This layer is a child of *Circle* , because as I said above, *Cylinder* has round faces, this round face is no different from the properties of a *Circle* , so *Circle* should be a basic class of *Cylinder* . Because *Cylinder* inherits the values from *Circle* , which *Circle* inherits from *Geometry* , so *Cylinder* has all the values of *Geometry* and *Circle* . It only needs to add the *High* attribute , and the *import* methods *enterHeight ()* , *calculateVolume ()* its own only.
- Similar for the relationships of *Rectangle* and *Square* . There is a special thing about the *Square* class , which is that since the long and wide sides of this shape are equal, the square's *enterSide ()* function just tells the user to enter an edge, then you assign the same edge value to the variable *height* and *width* , and so *Square* does not need to build any more methods, completely excellent inheritance from its parent class.

The results of running the program are as follows.

```
===== Circle ===== Radius =  
10  
Perimeter = 62.800003 Area = 314.0  
===== Cylindrical ===== Radius =  
10  
Height =  
2  
Volume = 628.0  
===== Rectangles ===== Length =  
10  
Width =  
2  
Perimeter = 24.0  
Area = 20.0  
===== Square ===== Side =  
5  
Perimeter = 20.0  
Area = 25.0
```

As for the called *InheritanceLearning* (*OOPLearning* and *PackageLearning* were closed). The way to organize classes into packages is as follows. project, I created a new project



The following is the source code of the corresponding classes in the program for your reference.

```
Class Geometry .
package shapes;
public class Geometry {
public final float PI = 3.14f;

public String name;
public float p;
public float area;
public float v;

public void printName() {
System.out.println("\n\n===== " + name + " =====");
}
public void printPerimeter() {
System.out.println("Perimeter = " + p);
}
public void printArea() {
System.out.println("Diện tích = " + area);
}
public void printVolume() {
System.out.println("Area = " + v); }
}

The Circle class .
package shapes;
import java.util.Scanner;
public class Circle extends Geometry {
public float radius;
// Constructor
public Circle() { name = "Circle"; }

public void enterRadius() {
System.out.println("Radius = ");
Scanner scanner = new Scanner(System.in); radius = scanner.nextFloat();

}

public void calculatePerimeter() { p = 2 * PI * radius;
```

```

}

public void calculateArea() { area = PI * radius * radius;
}

}

Class Cylinder .
package shapes; import java.util.Scanner;
public class Cylinder extends Circle {
public float height;
// Constructor
public Cylinder() { name = "Cylinder"; }
public void enterHeight() { enterRadius();

System.out.println("Height = ");
Scanner scanner = new Scanner(System.in); height = scanner.nextFloat();

}

public void calculateVolume() { calculateArea();
v = area * height;

}

}

Class Rectangle.
package shapes;
import java.util.Scanner;
public class Rectangle extends Geometry {
public float height; public float width;
// Constructor
public Rectangle() { name = "Rectangle";

public void enterHeight() {
System.out.println("Height = ");
Scanner scanner = new Scanner(System.in); height = scanner.nextFloat();

}

public void enterWidth() {
System.out.println("Width = ");

```

```
Scanner scanner = new Scanner(System.in); width = scanner.nextFloat();  
}
```

```
public void calculatePerimeter() { p = 2 * (height + width);  
}
```

```
public void calculateArea() { area = height * width;  
}
```

```
}
```

The *Square* class .

```
package shapes;  
import java.util.Scanner;  
public class Square extends Rectangle {  
// Constructor  
public Square() { name = "Square"; }
```

```
public void enterSide() {  
System.out.println("Side = ");  
Scanner scanner = new Scanner(System.in); height = width =  
scanner.nextFloat(); }
```

```
}
```

And finally the *MainClass* class .
package main;

```
import shapes.Rectangle; import shapes.Circle; import shapes.Cylinder; import  
shapes.Square;
```

```
public class MainClass {
```

```
public static void main(String[] args) { // Test with Circle class  
Circle circle = new Circle();  
circle.printName();  
circle.enterRadius();  
circle.calculatePerimeter();  
circle.calculateArea();  
circle.printPerimeter();  
circle.printArea();
```

```

// Test with Cylinder class
Cylinder cylinder = new Cylinder(); cylinder.printName();
cylinder.enterHeight();
cylinder.calculateVolume();
cylinder.printVolume();

// Test with Rectangle class Rectangle rect = new Rectangle(); rect.printName();
rect.enterHeight();
rect.enterWidth();
rect.calculatePerimeter();
rect.calculateArea();
rect.printPerimeter();
rect.printArea();

// Test with Square class Square square = new Square(); square.printName();
square.enterSide();
square.calculatePerimeter(); square.calculateArea();
square.printPerimeter(); square.printArea();

}

}

```

See, you have to code more and more, right? The main thing about learning to code is that you should not be afraid to code, although this's inheritance knowledge helps you a lot in terms of saving code lines, but not so that we programmers are completely free.

Lesson 19: this and super keywords

So you have just become familiar with to *inheritance* in Java from the previous lesson, through which you already know how to declare an inheritance relationship, when to inherit, and the inheritance property. What is the value from the parent class for the child class.

Coming to this's lesson, let's just talk about inheritance veto, but let's look at the definition and usage of the keywords *this* and *super*. They are quite important, but if you say too early, you cannot, because they are related to inheritance, if you say too late, you will not be able to understand some of the places to use them.

Let's get started.

I. This Keyword

Surely you remember. Keywords *this* was its use in *all 17*, then, is when you want to distinguish where the *variable* was where the *attributes* of the class, if they have the same name together. And in that lesson, I have not been clear about *this*. This section I will help you list all the uses that *this* brings.

As you know, the *this* keyword means: *is this object*. It references the object backwards, making it possible to access the values of that object. Here are a few useful roles of *this*, you will also see *this* used quite a lot in future Java lessons, and in *Android programming* as well.

1. Use *this* when accessing properties and methods in class

As you learned from *lesson 17*, when you used *this* to access a class's properties, the role of *this* is similar when you used *this* to access the method.

The main purpose of using *this* now as you know it is to help distinguish between variables and properties of the class, when they have the same name. However, you can absolutely use *this* anytime, anywhere when you want to access these properties and methods, like in the example below. But you also should not use *this* arbitrarily, it makes our code become cumbersome, as the example below is a bit overused *this*, I just want to show them a lot for you to see.

```
public class Circle { public float radius;  
// Constructor  
public Circle (float radius) { this.radius = radius; }  
  
public void calculatePerimeter () {  
// Calculate the circumference of the circle and print to the console  
}  
  
public void calculateArea () {  
// Calculate the area of a circle and print it to the console  
}  
  
public void printCircle () {
```

```
System.out.println ("Circle of radius =" + this.radius); this.calculatePerimeter ();  
this.calculateArea ();
```

```
}
```

```
}
```

2. Use this When Calling Another Constructor Inside Class

According to the constructor lesson, I should always talk about this usage of *this*, but I intentionally go to this's lesson in general.

If in one of your classes there are many constructors, and you want a certain constructor to call another constructor, often this type of call helps the constructors take advantage of each other's initialization code, avoiding rewriting.

this is used in a slightly different way than the one above, that is, you must call with an pass parameter like *this ([parameter])* , just like when you call a method. I call it *this ()* for easy presentation.

At this time, when calling, *this ()* will call a certain constructor, depending on its *pass_parameter* , the corresponding constructor is called. But calling another constructor does not create a new class, but just takes advantage of the constructor as when you call some function inside a class. You see an example.

```
public class Rectangle extendsHoc {
```

```
    public float height; public float width;  
    // Constructor  
    public () {  
        name = "Rectangle"; }  
  
    // Constructor  
    publicRectangle (float height, float width) { this (); // Call Rectangle ()  
        this.height = height;  
        this.width = width;  
    }  
  
    // Constructor  
    public (float) {  
        this (side, side); // Call Rectangle (height, width) }  
  
    public void calculatePerimeter () { p = 2 * (height + width);
```

```
}

public void calculateArea () { area = height * width;
}

}
```

Note that the *this()* keyword in this case is *only used in constructors*, to make use of other constructors like in the above example, if you put *this()* in normal methods. Otherwise, an error will occur from the system. One more thing, *this()* if any, then it must be the first line of code inside a constructor, in case below is false, the system will error because *this()* has other lines of code before.

```
// Constructor
public Rectangle(float height, float width) { this.height = height; this.width =
width; this(); }

}
```

3. Use *this* as the Parameter to Pass to Another Method or Constructor

In this part, through *Android programming*, you will use more. As for the framework of this Java lesson, it will be a bit difficult to find a good practical example, so I borrowed the code on the internet to show you how to use *this* keyword for parameter purposes this time. You will also find it easy to understand.

The following example is for using *this* as an argument to a method. The same is true of using *this* as a parameter passed to a constructor. Looking at the example, in class *Bar*, class *Foo* is declared as a parameter to the function *barMethod()* of class *Bar*. Then in the *Foo* class, where the line is highlighted, this class just passes in *barMethod()* via *this* keyword so the *Bar* class is allowed to use *Foo* as a fully declared class, i.e. you don't need to use the *new* keyword to create a class *Foo* any more.

```
public class Foo {
public void useBarMethod() { Bar theBar = new Bar(); theBar.barMethod(this);

}

public String getName() { return "Foo";
```

```
}

}

public class Bar {
    public void barMethod(Foo obj) { obj.getName();
}
}
```

4. Use this As An Expression of Returned Results

Where a result is *this* , it returns the instance of that class. This use of *this* can make you, and yourself, confused. Take a look at the following example before seeing what the confusion I want to talk about, and because of this confusion you may not need to use *this* in this case in practice. If you do know what *this* really means in this case, to help reduce confusion, please leave a comment for yourself.

First I have a *Student* class . As you can see in the *getStudent () method* , this class returns the *this* keyword .

```
public class Student {
    public String name; public String age;

    // Constructor
    public Student(String name, String age) { this.name = name;
        this.age = age;
    }

    public Student getStudent() { return this;
    }
}
```

Then, in the *main ()* function, there is a *Student* class declaration , see how *main ()* uses *getStudent ()* .

```
public class MainClass {
    public static void main(String[] args) {
        Student student = new Student("Yellow", "20");
        System.out.println("Name: " + student.getStudent().name);
        System.out.println("Age: " + student.getStudent().age);
    }
}
```

Actually the *getStudent ()* function in the *Student* class helps to return *this* ,

which is the current instance of the class, and so when you call `student.getStudent().Name`, it's actually still accessing the `name` property of the class. `Student`, and so the console output of the above code will be "Yellow" and "20" respectively. The real confusion happens, when it is possible to print the above output without the need for `getStudent()` like so. Notice the difference between the two `main()` functions.

```
public class MainClass {  
    public static void main(String[] args) {  
        Student student = new Student("Yellow", "20");  
        System.out.println("Name: " + student.name); System.out.println("Age: " +  
        student.age); }  
}
```

II. Keyword super

Unlike `this` keyword, it helps itself. The `super` keyword helps to `superclass`. That is, you can understand happily that `super` cannot be used to refer to the class "grandfather" okay.
to refer

refer to the

to the current object `superclass`, which is its closest

Here are some of the roles of `super`, you can compare the roles of `this` over the other to see the difference, and to make it easy to remember too.

1. Using `super` when accessing properties and methods of the nearest parent class

If the first purpose of `this` above is to distinguish between what is a variable and what is a property when they have the same name in a class. Then the first purpose of this `super` is to *distinguish what is the value of the subclass and what is the value of the closest superclass when they have the same name*. The name coincidence between the method of the child class and the parent class occurs more than the same name between the properties, and this name coincidence is intentional, you can preview *the negation lesson* for better understanding because Why has this same name.

The following example shows that the `Circle` class defines the same method `printName()` with its father, `Geometry`, and then `super` appears in the

printName () of *Circle* to help it call *printName ()* in the parent first, and then the remaining code. Again inside the *image ()* of *Circle*, you can test the code below.

```
public class Geometry { public String ten;
public void printName() {
System.out.println("\n\n===== " + name + " ====="); }
// Other methods here // ...
}
public class Circle extends Geometry {

// Constructor public Circle() { ten = "Circle"; }

public void printName() {
super.printName();
System.out.println("\n Method from Circle");

}
// Other methods here // ...
}
public class MainClass {

public static void main(String[] args) { // Little test with Circle
Circle circle = new Circle();
circle.printName();

}
} 2. Using super When Calling A Constructor Of The Nearest Parent Class
```

In the same way when you use *this ([transfer_parameter])*, *super ([transfer parameter])* helps you call, and takes advantage of the initialization code in the nearest parent class.

And just like the keyword *this ()*, *super ()* is only used in constructors, if you put this *super ()* in other normal methods, an error will occur from the system. And similarly, if the keyword *super ()* is present, it must be declared first inside a constructor.

```
public class Geometry {
public String name;
```

```
// Constructor
public Geometry(String name) { this.name = name;
}
// Other methods here // ...
}
public class Circle extends Geometry {
// Constructor public Circle() { super("Circle"); }
// Other methods here // ...
}
```

So we have just finished looking at the concepts and use case of the two keywords *this* and *super*. They are quite important and appear a lot in our projects or examples from now on, please slowly master and practice using them. Of course, this's lesson only has theory, practices related to these two keywords will appear in the following lessons.

Lesson 20: Getting Started With Inheritance

So we have passed through many important knowledge in object oriented programming, such as *properties* , *methods* , *constructors* . But there is one kind of knowledge that can be said to be the quintessence of object oriented, which we will approach starting in this's lesson that will give you a way to use and organize classes in your application in such a way. The knowledge is completely more advanced and effective than the ways that you have been familiar with in previous lessons, which is the knowledge of Inheritance.

If you are new to Java and object oriented, inheritance will make it a little more difficult. At this point, Java code is no longer as important as how you organize the structure for classes or objects within your application. In my experience, after getting acquainted to the inheritance, there will be countless questions to bring to you. Yes, you will wonder when to inherit. There are also friends who understand inheritance, but the organization of inheritance wwidthly "*relatives*" of them, making the inheritance become more confusing. Then when should we block the inheritance of any given class? etc. These questions I will try to cover thoroughly in this series of lessons on inheritance, so that you have a best way to apply inheritance to your product.

I. Familiar With Inheritance

1. What Is Inheritance?

Inheritance in object-oriented programming refers to a relationship between objects, some say this relationship is *Father-Con*, some say it is an *Extended* relationship. People seem to like the concept of *Father-Son* more, but I see *Openness*, and most closely, it is better to understand that *Inheritance* is always better. Because it is essentially a reuse, in some cases it extends the properties of an object from some other object.

So, to understand a practical way, suppose we first have a certain class, this class can be written by us, or "*gleaned*" somewhere, we temporarily name this available class. is *A*. Then, in order to take advantage of *A*'s functions or properties without having to rewrite (or copy over, maybe piracy), we build a new class that inherits from *A*, we call the class. this new *B*. Then our *B* will have available methods and properties that *A* has. There are times when it is not for the purpose of *reusing A*'s values, but because some of *A*'s values do not match *B*'s needs., That is the inheritance from *A* also helps *B* have the opportunity to complete the (also known as extensions) values were not right of *A* that does not alter the nature of the *A*. This, we are focusing on *Reuse*, *Expansion* purposes, we will talk about in the next lesson.

2. Why Must Inherit?

Through my above ideas, perhaps you also understand the reasons why they must inherit right.

Well, *the main purpose of inheritance is to reuse, and to further extend, properties and methods available from an object*.

That's all, the purpose of inheritance is not sublime at all, but the effect it brings is very great. By reusing these existing ones, your project structure will look more professional and your code will be easier to read. In addition, inheritance also helps you reduce the burden of coding a lot, because you take advantage of the existing code of other classes.

II. Inheritance In Java

We are talking about concepts in general. So how do we demonstrate inheritance in Java? In Java, to express a class that you want to inherit from a certain class, you use the *extends* keyword .

Take a look at the example below, you should not rush to code, later in the practice we will code together.

```
class Circle {  
    float r;  
    float getRadius() {  
  
        return r;  
    }  
}  
  
class Cylinder extends Circle {  
}
```

The above code is an explicit representation of inheritance, where the *Cylinder* class inherits *Circle* with the keyword *extends*. And according to the law of inheritance, *Cylinder* can inherit the values (properties and methods) that *Circle* has declared. Through such inheritance relationship, one can call the *Circle* class *the Base Class*, or the *Parent Class*. And *Cylinder* class is called *Derived Class (Derived Class)* or class *Child (Subclass, Child Class)*.

Usually, the superclass, or base class, is the class that holds the common values, or the most base values, for the subclasses. So if you have multiple layers with certain similarities, like the Circle and the Cylinder in the example above, these layers all have round faces (cylinders have two round faces), so you can use Circle as the layer. base (since it contains the minimum values that the Cylindrical can take advantage of, in this case the round face itself). Or there are cases where there are many classes with similar values, in which you can combine into a single base class and then the subclasses just inherit and reuse those similar values without Need to declare something more, as in the exercise below, I will create a class *Geometry* is the most basic for the classes *Circle*, *Square*, *Rectangle*, ...

Above is an example of when you need to inherit, but now I would like to list some important points in the process of organizing your inheritance.

- A class is only allowed to inherit from one and only one parent class .

- Although not inherited from many superclasses, but the parent class that this inheriting object can inherit from another superclass, you can call this other *superclass* as "*grandfather class*" for easy remembering, and surely there could

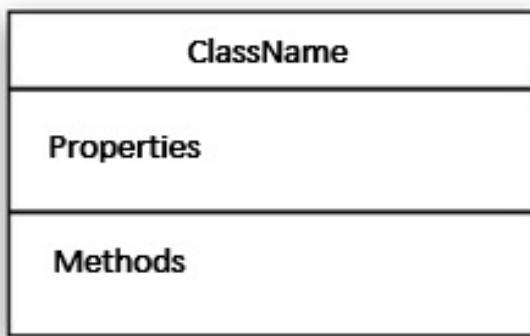
be a "*great-grandfather class*" if the "*grandfather class*" inherits another class.

- If a class does not declare inheritance at all (like the classes we have practiced in the previous lessons), then the system will default to see it inheriting from the *Object* class . In the next lesson, we will talk about this *Object* class .

III. Familiarize With Class Diagrams

The concept of inheritance in this's lesson is just that. But before going into specific practice, I invite you to familiarize yourself with a divine diagram, which if you are a mainstream programmer, you cannot ignore. The main thing about this section is to talk about how you see and understand the diagrams. Because our project becomes more and more complex, there will be many and many classes, they use each other, inherit each other. And so without a diagram, we would not be able to describe all these relationships in words.

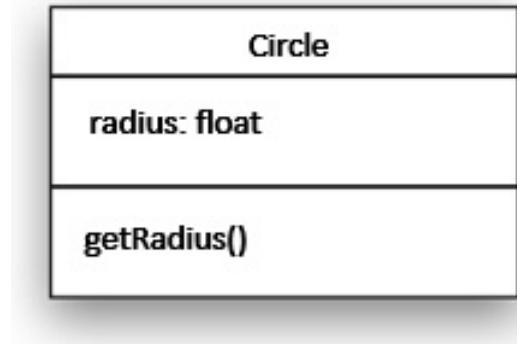
The class diagram of this section was taken from the *UML* 's *Class Diagram* construction principle . UML is a set of many different principles for software system specification and design. Say that so that you understand this is a standard principles compliance diagram, if you understand and follow these principles, you will create a common speaking model, which everyone who reads and understands you want Shows what to your software. You know, you are a little bit familiar with this diagram, because I used it in *lesson 16* , then I want to express a class with three main components as follows, and the shapes you see is exactly how the class diagram represents a class instance.



The shape and color of the blocks in the class diagram may differ in our lesson and in other documents, depending on the tool for drawing it. But even though they differ in appearance, in general, the data that each diagram shows must

comply with a cube principle with the above three components.

I take the example of the *Circle* class like the code above, then when represented as an entity in the class diagram, it looks like this.

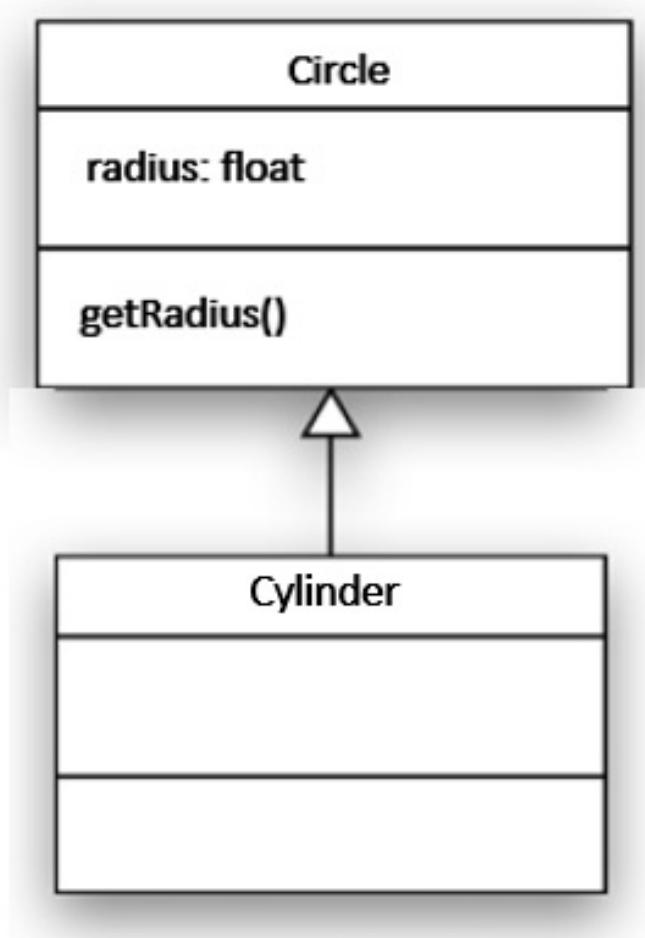


Looking at the diagram, you know right away that you need to build a class named *Circle* , this class has a property named *r* with data type *float* , and a *getRadius () method* returns *float* . Although it has a static value, it does not show the content or relationship of the elements inside a class, but

surely you can easily understand and visualize how to build a class based on a diagram. This is how.

Next, I continue to talk about how this class diagram shows inheritance. With the expectation that the *Cylinder* class *would* inherit from *Circle* , we can show through this relationship class diagram with an arrow mark as follows. You must pay attention to the empty arrow mark as shown, if you use a different type of arrow, it will cause confusion with other relationships.

One more idea of the diagram, for example in the case in *lesson 18* , the *Circle* class uses the *Point* class to make a property, which is named *prescriptionDo* , so the diagram will show this usage as follows: .



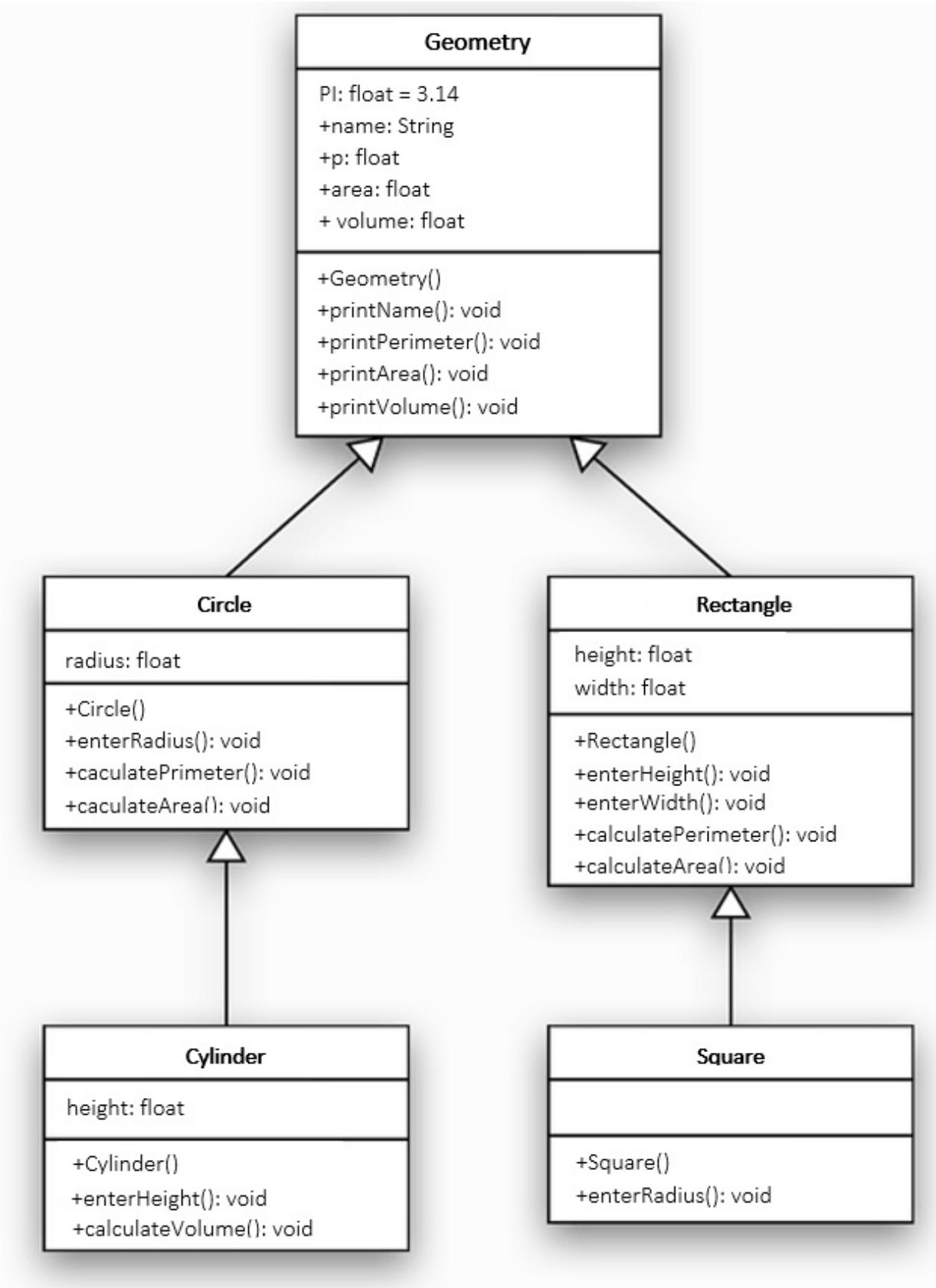
Everything is really clear, isn't it? Done, with this's lesson, I only briefly presented about inheritance and class diagram like that. We will add knowledge and symbols to this diagram in the next lessons. Now we need to practice getting used to.

- *Inheritance Practice*

We will continue to build applications to calculate geometrical values for *Circle* , *Cylinder* , *Rectangle* , and *Square* . You should also know that, if you do not apply the inheritance knowledge of this's lesson, you will still build the results of this exercise perfectly with the knowledge of OOP in the previous lessons. can try. But with the application of inheritance, as I said, you will save significant lines of code. Then invite you to try building a new project with me.

First of all, I invite you to look at the class diagram of this's lesson (with

inheritance applied) as follows, please consider a little bit (note the + signs in front of each method or property in the diagram are declared with the keyword *public* , this is *accessible* to the class values, which we will talk about later).



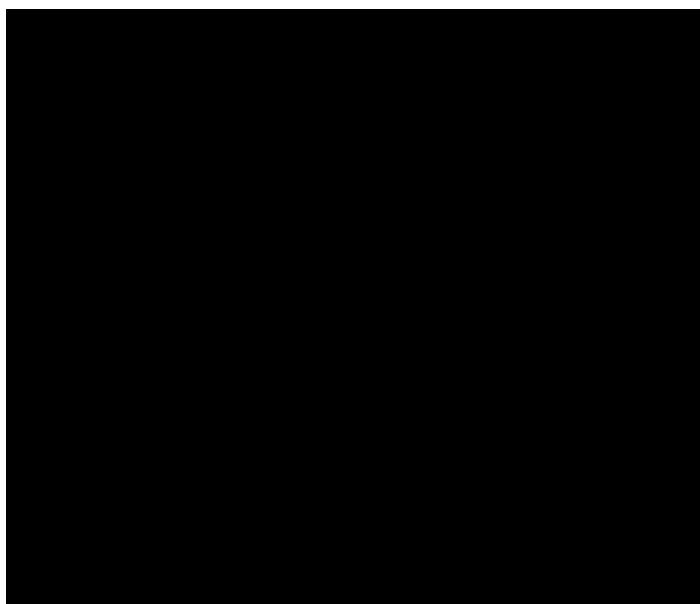
Before going into the official, I will look at the diagram and detail each class, then show you the results of running the program, and finally the code of each class. Please take a look at the code of the classes quickly, but rely on the diagrams and results and try to code, each person will have a different way of code, you do not have to code the same as me, as long as your program running fine is fine.

- Grade *Geometry* . This is the superclass of the remaining classes, or means the most basic class. Since it is a base class, it would be best if it contained properties or methods that would be useful to subclasses, or it could be said that the subclass could effectively inherit those values. For example, the *PI* constant I will declare in this class. The *name* attribute is shared, but will be specifically defined by subclasses according to their name. The *p* , *area* , *v* are the same, although they are defined in general, the subclasses will contain different values. The methods of this superclass are also meant to be used by subclasses, so they will have a specific function body, like *printName ()*. will output *name* out console. Or *printPerimeter ()* , *printArea ()* , *printVolume ()* will also output the variables *p* , *area* , *v* respectively.
- *Circle* class . A subclass of *Geometry* . As you know *Circle* will inherit the values from its superclass. In addition to the properties it gets from the superclass *name* , *p* , *area* , *v* , it also defines its own special attribute *radius* . You can initialize the variable *name* for *Circle* in the *Circle ()* constructor . The methods *enterRadius ()* , *calculatePerimeter ()* , *calculateArea ()* are not declared in the parent class, *Circle* designs itself.
- *Cylinder* class . This layer is a child of *Circle* , because as I said above, *Cylinder* has round faces, this round face is no different from the properties of a *Circle* , so *Circle* should be a basic class of *Cylinder* . Because *Cylinder* inherits the values from *Circle* , which *Circle* inherits from *Geometry* , so *Cylinder* has all the values of *Geometry* and *Circle* . It only needs to add the *High* attribute , and the *import* methods *enterHeight ()* , *calculateVolume ()* its own only.
- Similar for the relationships of *Rectangle* and *Square* . There is a special thing about the *Square* class , which is that since the long and wide sides of this shape are equal, the square's *enterSide ()* function just tells the user to enter an edge, then you assign the same edge value to the variable *height* and *width* , and so *Square* does not need to build any more methods, completely excellent inheritance from its parent class.

The results of running the program are as follows.

```
===== Circle ===== Radius =  
10  
Perimeter = 62.800003 Area = 314.0  
===== Cylindrical ===== Radius =  
10  
Height =  
2  
Volume = 628.0  
===== Rectangles ===== Length =  
10  
Width =  
2  
Perimeter = 24.0  
Area = 20.0  
===== Square ===== Side =  
5  
Perimeter = 20.0  
Area = 25.0
```

As for the called *InheritanceLearning* (*OOPLearning* and *PackageLearning* were closed). The way to organize classes into packages is as follows. project, I created a new project



The following is the source code of the corresponding classes in the program for your reference.

```
Class Geometry .
package shapes;
public class Geometry {
public final float PI = 3.14f;

public String name;
public float p;
public float area;
public float v;

public void printName() {
System.out.println("\n\n===== " + name + " =====");
}
public void printPerimeter() {
System.out.println("Perimeter = " + p);
}
public void printArea() {
System.out.println("Diện tích = " + area);
}
public void printVolume() {
System.out.println("Area = " + v); }
}

The Circle class .
package shapes;
import java.util.Scanner;
public class Circle extends Geometry {
public float radius;
// Constructor
public Circle() { name = "Circle"; }

public void enterRadius() {
System.out.println("Radius = ");
Scanner scanner = new Scanner(System.in); radius = scanner.nextFloat();

}

public void calculatePerimeter() { p = 2 * PI * radius;
```

```

}

public void calculateArea() { area = PI * radius * radius;
}

}

Class Cylinder .
package shapes; import java.util.Scanner;
public class Cylinder extends Circle {
public float height;
// Constructor
public Cylinder() { name = "Cylinder"; }
public void enterHeight() { enterRadius();

System.out.println("Height = ");
Scanner scanner = new Scanner(System.in); height = scanner.nextFloat();

}

public void calculateVolume() { calculateArea();
v = area * height;

}

}

Class Rectangle.
package shapes;
import java.util.Scanner;
public class Rectangle extends Geometry {
public float height; public float width;
// Constructor
public Rectangle() { name = "Rectangle";

public void enterHeight() {
System.out.println("Height = ");
Scanner scanner = new Scanner(System.in); height = scanner.nextFloat();

}

public void enterWidth() {
System.out.println("Width = ");

```

```
Scanner scanner = new Scanner(System.in); width = scanner.nextFloat();  
}
```

```
public void calculatePerimeter() { p = 2 * (height + width);  
}
```

```
public void calculateArea() { area = height * width;  
}
```

```
}
```

The *Square* class .

```
package shapes;  
import java.util.Scanner;  
public class Square extends Rectangle {  
// Constructor  
public Square() { name = "Square"; }
```

```
public void enterSide() {  
System.out.println("Side = ");  
Scanner scanner = new Scanner(System.in); height = width =  
scanner.nextFloat(); }
```

```
}
```

And finally the *MainClass* class .
package main;

```
import shapes.Rectangle; import shapes.Circle; import shapes.Cylinder; import  
shapes.Square;
```

```
public class MainClass {
```

```
public static void main(String[] args) { // Test with Circle class  
Circle circle = new Circle();  
circle.printName();  
circle.enterRadius();  
circle.calculatePerimeter();  
circle.calculateArea();  
circle.printPerimeter();  
circle.printArea();
```

```

// Test with Cylinder class
Cylinder cylinder = new Cylinder(); cylinder.printName();
cylinder.enterHeight();
cylinder.calculateVolume();
cylinder.printVolume();

// Test with Rectangle class Rectangle rect = new Rectangle(); rect.printName();
rect.enterHeight();
rect.enterWidth();
rect.calculatePerimeter();
rect.calculateArea();
rect.printPerimeter();
rect.printArea();

// Test with Square class Square square = new Square(); square.printName();
square.enterSide();
square.calculatePerimeter(); square.calculateArea();
square.printPerimeter(); square.printArea();

}
}

```

See, you have to code more and more, right? The main thing about learning to code is that you should not be afraid to code, although this's inheritance knowledge helps you a lot in terms of saving code lines, but not so that we programmers are completely free.

Lesson 21: Overriding In Inheritance

With this's lesson, I will add the next knowledge in the knowledge of *Inheritance* . If in *lesson 21* , you already know how to use the *extends* keyword to express the *Inheritance* from one class to another. And then, you also get used to reuse all the values from a *Parent* class (or *Base* class) to the *Child* class (or *Derivative* class), at that time I call this *Reuse in Inheritance* . This we are going to take a look at the next aspect of inheritance, which no longer makes sense. *Reuse* , it is *Overriding* .

I. What Is Overriding?

Veto is also a characteristic of Inheritance . I used to use the word *veto* , maybe because of the habit from when I used C ++ or C # . When I came to the Java language, I found it called *Override* .

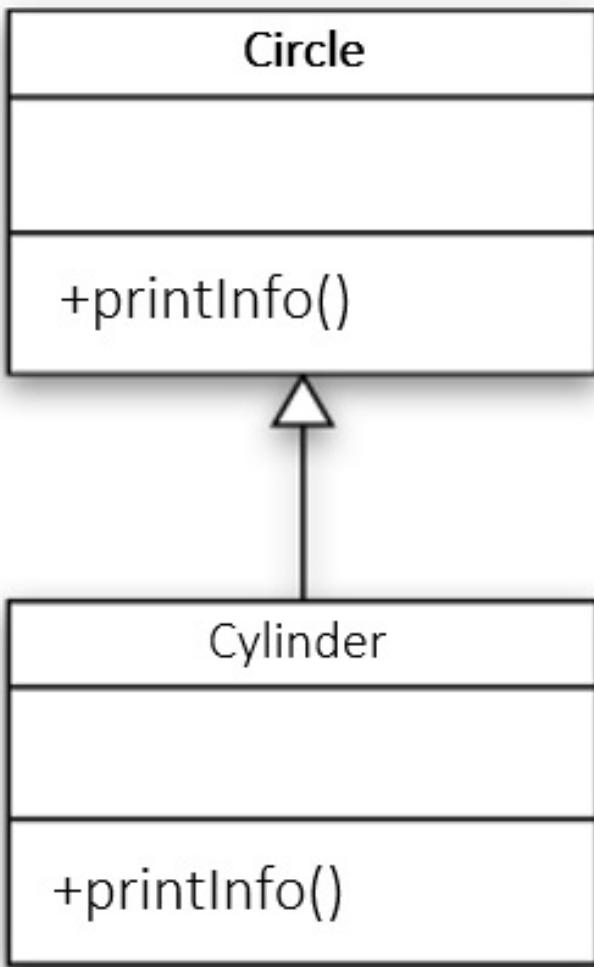
As I have mentioned before, if you are familiar with *Inheritance* , you know that this concept implies characteristics *Reusing* , we looked briefly at *lesson 21* , this feature allows subclasses can by default get the methods and properties that its parent class has declared (and allowed).

What about the *Veto* feature ? This property allows the subclass to declare methods identical to the methods of the parent class, and then the subclass will redefine the content of that method, which means "*disapprove*" of the method. That consciousness of the parent class, because of this meaning that people often call *Veto* (or *Overriding* , or *Overriding*) is so.

II. What is Overriding?

As mentioned above, *for overriding needs, you just need to name the method in the child class with the same name and pass parameters with the method already in the parent class* . Then, it's not required, but you should declare an annotation (called *Annotation* in English) with @*Override* content on each overriding method for code clarity. With this overridden method in the subclass, you can redesign the logic inside it, once in a place calling that subclass's method, instead of using the parent class method according to the inheritance principle. , then the method of the subclass is used.

Let's see the following example. The example will use two classes *Circle* and *Cylinder* , where *Cylinder* inherits from *Circle* . Let's try declaring the *printInfo* () method in both classes, following the class diagram below.



Their code is as simple as follows.

```
public class Circle {  
    public void printInfo() {  
  
        System.out.println("This is Circle");  
    }  
}  
  
public class Cylinder extends Circle {  
  
    @Override  
    public void printInfo() {  
        System.out.println("This is Cylinder");  
    }  
}
```

```
}
```

With the declaration of two classes as above, then you can see how to call them in *main ()* function .

```
public class MainClass {
```

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

```
        Circle Circle = new Circle();
```

```
        Cylinder Cylinder = new Cylinder();
```

```
        Circle.printInfo(); Cylinder.printInfo(); }
```

```
}
```

If you execute the program, you will see two commands *printInfo ()* in two classes that will print to the console as follows.

```
This is Circle
```

```
This is Cylinder
```

That is because *printInfo ()* in *Circle* has been overridden (or vetoed, overwritten) by *printInfo ()* in *Cylinder* . If you experiment by deleting *printInfo ()* from *Cylinder* , you will get the same two lines that print to the console, which is because without this override, the reuse of inheritance will take advantage. You already understand the concept of overriding, right.

There's an open mind for you. That is, if you want the overriding method to use the duplicate method of the parent class, then you can use *the super keyword* as in the example below.

```
public class Cylinder extends Circle {
```

```
    @Override
```

```
    public void printInfo() {
```

```
        super.printInfo();
```

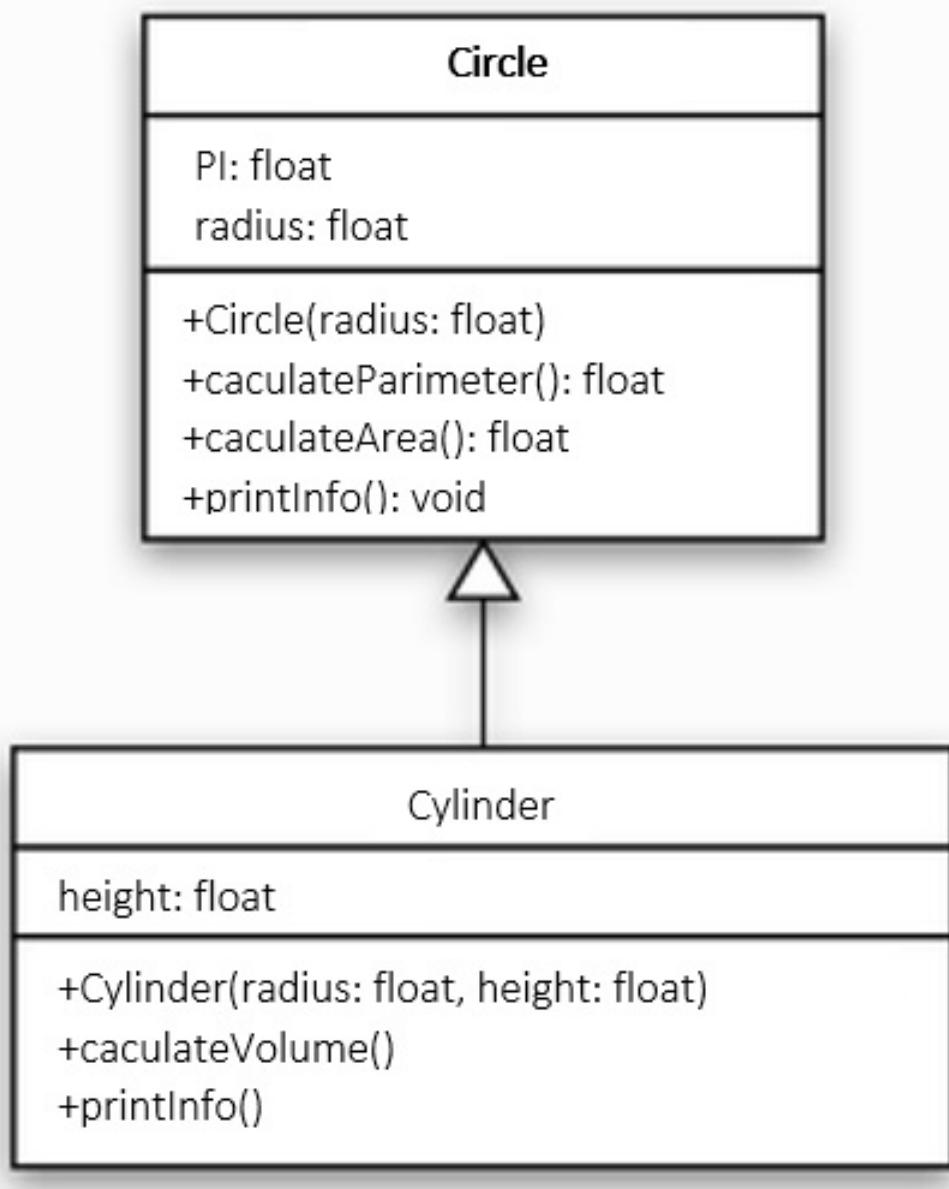
```
        System.out.println("This is Cylinder"); }
```

```
}
```

• Inheritance Practice (Negative)

The theory of overriding is only as above. Easy to remember, right. Now let us try to create a complete relationship between the two layers *Circle* and *Cylinder* . It's still inheritance, of course, but you'll see the full range of *Reuse* and *Veto features* .

You can create a new project for this exercise.
Let's take a look at the following class diagram.



In the *Circle* class , we do not need the method to ask the user to enter the radius from the console like in the previous exercises, but pass this radius to the constructor immediately. The *calculateParimeter ()* and *calculateArea ()* methods both return calculation results based on the radius obtained from the constructor. And finally, the *printInfo ()* method will be like the above example, this method will be overridden by the inheriting *Cylinder* class .
public class Circle {

```

public final float PI = 3.14f;
public float radius;
// Constructor
public Circle(float radius) { this.radius = radius; }

public float calculateParimeter() { return 2 * PI * radius;
}

public float calculateArea() { return PI * radius * radius;
}

public void printInfo() {
System.out.println("This is Circle");
System.out.println("Circle has Parimeter: " + calculateParimeter() +
"and Area: " + calculateArea()); }
}

```

In the *Cylinder* class , this class inherits the *PI* and *radius properties* , the *calculateParimeter ()* and *calculateArea () methods* . *Cylinder* additionally declares the attribute *height* and the method *calculateVolume ()* . And finally, the *printInfo () method* will override the parent class.

```

public class Cylinder extends Circle {
public float height;

// Constructor
public Cylinder(float radius, float height) { super(radius);
this.height = height;
}
public float calculateVolume() {
return calculateArea() * height; }

@Override
public void printInfo() {
System.out.println("This is Cylinder");
System.out.println("Cylinder has Volume: " + calculateVolume()); }
}

```

Accomplished. Now we will use them in the *main ()* function .

```
public class MainClass {  
    public static void main(String[] args) { Circle Circle = new Circle(10);  
        Cylinder Cylinder = new Cylinder(10, 20);  
        Circle.printInfo(); Cylinder.printInfo(); }  
}
```

And here are the results of launching the program.

This is Circle

Circle has Parimeter: 62.800003 and Area: 314.0

This is Cylinder

Cylinder has Volume: 6280.0

We have just seen one more feature in the OOP knowledge series. With this knowledge of overriding, I see it as part of the *Inheritance* , it talks about the *Veto* , besides the *Reuse* that we have become familiar with from the previous lesson.

Lesson 22: Object class

This, we continue to talk more deeply about OOP, especially about Inheritance . Let's review a bit that, if someday you just approach inheritance , then you get used to the veto, or override in inheritance , then this, you will get acquainted with a class, called the *Object* class , to see how this class affects the use of classes or objects that you are already familiar with.

I. What is Object class?

The appearance of the *Object* class at this point can be a little confusing. But when you just got into Java, in your first lines of code, this class has already appeared. *This class is named Object, it is a class made available by the system, and the system also designates it as the highest superclass of all classes in Java .*

For better understanding, let's review the beautiful *Circle* class *declaration* from the previous lessons.

```
public class Circle {  
}
```

You see that this *Circle* has no inheritance from any other class, and you think it

is the highest parent, if there is a declaration of other classes *extends* from it? No, as I mentioned above, *in Java, if you do not specify a superclass for a certain class, the default system for that class inherits from a class, which is the Object class*. But because it is the default inherited by other classes (except for the class declared inheriting to another class, because you already know that *a class cannot have many superclasses*), you don't have to declare it. tell explicitly what the inheritance to this *Object* class does.

The code below I want to write to it explicitly, let you know that the declaration like this is implicit, no one writes like that.

```
public class Circle extends Object { }
```

II. What Does the Object Class Do?

Well surely you will wonder, then what the hell is born of this hell, so that it can be a father of another?

In fact, if you read through my mind below, you will see that the *Object* class is quite important.

Firstly, *Object* can be used for you to declare an object in advance but you do not know what that object is, and after in specific situations, you will force this type of temporary *Object* class to classes. corresponding specific child. It sounds vague, but keep in mind, you will understand this benefit in the lessons to come.

Second, as you already know the main use of inheritance, it is the use of helping to group similar values of classes into the same base class, like the example of a class *Geometry* is the base class. for the classes *Circle* , *Rectangle* , *Square* , ... in *lesson 21* . So is the *Object* class , this class now acts as a base class for any class you create in a Java project. So this base class here will contain the most useful common values for the rest of the class, please continue to see.

III. Methods The Object Class Provides

The methods below of the *Object* class have no immediate practical meaning, that is, you just read it and leave it there, you can use it later. You should remember that these methods are available to all classes by *Object* , so this you read briefly, and then remember, later on there is a chance to use it, do not redesign it yourself. Please.

1. public final Class getClass ()

This method returns a class *Class* , class *Class* contain the information available relating to the construction objects are called.

You can refer to the following code, you see, we do not declare these methods, it is available in the *Object* class and we just need to get it to use it.

```
public static void main(String[] args) {  
    Circle Circle = new Circle();  
    System.out.println("Infomation of object Circle: " + Circle.getClass());  
    System.out.println("Infomation of object Circle: " +  
        Circle.getClass().getName()); System.out.println("Infomation of object Circle: " +  
        Circle.getClass().getSimpleName());  
}
```

The result prints some useful information about this object, depending on the method you invoke deeply in the *Class* .

```
Information of object Circle: class shapes.Circle Information of object Circle: shapes.Circle Information of  
object Circle: Circle
```

2. int hashCode ()

This method returns a *hash code* value . I have never used this method before, but I think it can be used as values that distinguish objects from each other.

As the code below, you can see when printing the hash code of two objects of the same class.

```
public static void main(String[] args) { Circle circle1 = new Circle(); Circle  
Circle2 = new Circle();
```

```
System.out.println("Hashcode of Circle 1: " + circle1.hashCode());  
System.out.println("Hashcode of Circle 2: " + Circle2.hashCode()); }
```

```
Hashcode of Circle 1: 1735600054 Hashcode of Circle2: 21685669
```

3. boolean equals (Object obj)

Is this *equals ()* method familiar? No? Remember it.

Actually, in *lesson 13* , lesson about *String*, I mentioned this *equals ()* method in *String* which helps compare two *String*'s the same or not. By now you know that *String is also an Object, and so it will also inherit this equals () method from the superclass Object* . But more interestingly, it's that *equals ()* of *String* had overridden this method of the parent class, not completely inherited.

So similar to String, this *equals ()* method helps compare any two objects with each other to see if they are the same. Two objects are said to be identical, not because they are created from the same class, but because they have the same reference value. This reference problem is actually related to pointer knowledge, but Java doesn't want its programmers to know about pointers, so I will talk about the concept of references later. .

Surely you can also guess the return results of the two following comparison methods, right?

```
public static void main(String[] args) { Circle circle1 = new Circle(); Circle  
Circle2 = new Circle();
```

```
System.out.println(circle1.equals(Circle2)); }
```

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

```
Circle circle1 = new Circle();
```

```
Circle Circle2 = circle1;
```

```
System.out.println(circle1.equals(Circle2)); }
```

You can also print out the hash code between them so you can see the relationship between *equals ()* and *hashCode ()* .

4. Object *clone ()*

This method initializes and returns a copy of the called object. In essence, this method I just mentioned without a specific example for it, because currently the *Object* class is declaring the *accessibility* of this method is *protected* . You will understand this type of accessibility later, but it's kind of protected and only useful when its subclass overrides this method.

5. String *toString ()*

This method helps to return a String type of expression for this object. Its content is a combination of string (*getClass ()* . *GetName ()* + "@" + *Integer.toHexString (hashCode ())*).

Interestingly, *this method may not need to be called explicitly* . That is why the String objects that you are familiar with do not need any *toString () statements* , as the following example shows the program prints two objects, a *String* , plus an *imageTron1* .

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

```
Circle circle1 = new Circle();
System.out.println("Don't call toString() explicitly:" + circle1);

}
```

It is the equivalent of explicitly calling `toString()` like this, but you don't need to.
public static void main(String[] args) { Circle circle1 = new Circle();
System.out.println("Call toString() explicitly: ".toString() + circle1.toString()); }

Their execution results are the same.

Don't call `toString()` explicitly: [shapes.Circle@677327b6](#)

Call `toString()` explicitly: `circle1.toString()`

6. `void finalize()`

Object subclasses can override this method, so that actions can be used to free up the resources in use, before the Garbage Collection - I mentioned this garbage collection. *Opening* bit) clean up this unused object.

You also saw that this method is automatically called by the system. And it's also quite indepth, sometimes used when you are using it and want to free up resources related to reading files, or networking, which we will become familiar with in the following lessons. In addition to the above useful methods that the *Object* class brings to its subclasses, there are a number of other methods, such as `notify()`, `notifyAll()`, `wait()`, which will be mentioned in the lessons. are related to them later.

Lesson 23: Acessibility (Access Modifier)

This we are going to talk about a problem that has been promised for a long time, since you have just become familiar with the *properties* and *methods* of the class, which is the knowledge of the *ability to access class values*. So what are these *Accessibility* and how important are they, please watch the lesson.

I. What Is Access Modifier?

Accessibility, also known as *Access Modifier*, are Java provided *Definitions*. And you, or other programmers, will use these *Definitions* to express *permissions to access class values*. The *values of the class* here I want to mention include *properties*, *methods*, and *constructors* of a class.

The *Access Modifiers* in Java are the keywords: *private* , *protected* , *public* and *default* (*meaning nothing is defined*) .

II. What is the Definition of Accessibility?

Let's take a look at each of the *accessibility capabilities* mentioned above, but let's review how to declare one.

Remember, in the syntax related to the declaration of class values, I have stated how to declare this *Accessibility* .

For example, when declaring the *properties of this class* .

[acessibility] attribute_type_type [= initial_value];

Or when declaring the *method of this class* .

[acessibility] payload type () {

// Lines of code

}

Or when declaring this *constructor* . [accessibility] method name () {

// Lines of code

}

So you can understand how to define an *accessibility* class values, right? Below is a complete example of the *Accessibility* declaration in the *Circle* class .

```
public class Circle {  
    protected final float PI = 3.14f; private float radius;  
    // Constructor  
    public Circle(float radius) { this.radius = radius; }  
  
    protected float calculatePerimeter() { return 2 * PI * radius;  
    }  
  
    protected float calculateArea() { return PI * radius * radius;  
    }  
  
    public void xuatThongTin() {  
        System.out.println("This is Circle");  
    }  
}
```

III. Meaning Of Accessibility

Now, let's go through each of the *Accessibility* to see what they mean and do.

First of all, I would like to summarize the *Accessibility* based on a table below for easy recall. This table simply carries the values “Yes” if the definition is accessible at some range, and carries the value “No” to the opposite meaning.
private default protected public

Inside the classroom

Yes Yes Yes Yes

In another class, but in the same package

No Yes Yes Yes

In the subclass, in the same package No Yes Yes Yes

In a subclass, different from package

No No Yes Yes

In any other class (not subclass), other than package No No No Yes

Table of meanings of accessibility

Now I will elaborate on each *Accessibility* .

1. *Private Accessibility*

If you specify a certain value (property, method or constructor) of the class to be *private* , as in the table above, you can only access this value within that class.

This *private* ability is meant to protect the values inside the class from being “*visible*” (read and edited) by the outer classes .

- If *private* is specified for a property , that property will not be allowed to access or modify from other classes, unless you build *getter* and *setter methods* for that property, which we'll talk about later.
- Even if *private* is assigned to a method , that method will not be accessed or inherit from other classes.
- And if *private* is specified for the constructor , this constructor cannot be used to instantiate the object of that class.

As far as I know, this *private* ability is used a lot, if you are not sure whether a value should be called or reused, then just specify *private* for it.

- Practice # 1

In this exercise, we create a class *Circle* that declares the values to be *private* .

You can see that these values when used inside the *Circle* class are very good, with no errors from the system.

```
public class Circle {  
    private final float PI = 3.14f; private float radius;  
  
    // Constructor private Circle() {}  
  
    private float calculatePerimeter() { return 2 * PI * radius;  
    }  
  
    private float calculateArea() { return PI * radius * radius;  
    }  
}
```

So you can test for yourself, with the call to these values of *Circle* from *main()* function , the following line of code will be error.

```
public class MainClass {  
  
    public static void main(String[] args) { Circle circle = new Circle();  
    circle.radius = 20;  
    circle.calculateArea();  
  
    }  
}
```

The result line shows the error below.

Line 4 error

Line 5 error

Line 6 error

2. Accessibility default

As I said, the *default* here means *no accessibility definition at all* , it's like the introductory lessons we've been familiar with, then we temporarily leave the *Accessibility* declarations blank. This *update* .

And with the *accessibility* of class values blank , you can see that, if within that class or other classes in the same package, it will be able to see and access those values. Visibility and access are only blocked when you call these values from a

class outside the package.

This ability is less commonly used, because everyone wants transparency in code.

- Practice # 2

Here I leave out all the *Accessibility* declarations for the values of the *Circle* class .

```
package shapes;  
public class Circle {  
    final float PI = 3.14f; float radius;  
  
    // Constructor Circle()  
}
```

```
float calculatePerimeter() { return 2 * PI * radius; float calculateArea() {  
    return PI * radius * radius; }  
}
```

And similarly, with the following lines of code declared in the *main ()* function , you can guess which line will fail.

```
package main;  
import shapes.Circle;  
public class MainClass {  
  
    public static void main(String[] args) { Circle circle = new Circle();  
    circle.radius = 20;  
    circle.calculateArea();  
  
    }  
}
```

The error line result is as follows.

Line 8 error Line 9 error Line 10 error

The reason these lines are error is because they are in package "main" which is different from "shapes" package of Circle. You try to bring them back to the same package and verify.*Protected Access*

This capability is meant to protect class values for reuse or *override* of its subclasses . Therefore, *protected* capabilities will grant complete freedom to subclasses, whether in the same or different packages. This ability is limited to

classes that are not subclasses and are outside the package.

- Exercise Number 3 This time look at the *protected* declarations for the values inside *Circle* as follows.

```
package shapes;
public class Circle {
    protected final float PI = 3.14f; protected float radius;
    // Constructor
    public Circle(float radius) { this.radius = radius; }

    protected float calculatePerimeter() { return 2 * PI * radius;
}

    protected float calculateArea() { return PI * radius * radius;
}
}
```

And with the code in the *main ()* function , again try to guess which line of code gets an error.

```
package main;
import shapes.Circle;
public class MainClass {

    public static void main(String[] args) { Circle circle = new Circle(10);
circle.calculateArea();

}
```

Line 9 has an error

caculateArea() is the only “NO” case of protected.

Cause it was call from a class that don’t have any parent-child ralationship with Circle class and also stay at different package with Circle

Exercise Number 4

Still with the code of *Exercise 3* , this time I declare more *Cylinder* inherits from *Circle* , and the *main ()* function as follows. You see if there is any error message.

```
package shapes;
public class Cylinder extends Circle {
    public float height;

    // Constructor
```

```

public Cylinder(float radius, float height) { super(radius);
this.height = height;
}

public float calculateTheTich() { return calculateArea() * height;
}
}

package main;

import shapes.Cylinder;
public class MainClass {

public static void main(String[] args) { Cylinder cylinder = new Cylinder(10,
20); cylinder.calculateTheTich();
This is the result of the error lines.

```

No line has error

Because Cylinder has inherited methods from super class and
reassign accessibility of these methods

3. Public Access

Perhaps it doesn't take a lot of paper to talk about this possibility. If you specify a value in the class to be *public* , then it is assumed that all other classes have access to these values.

So I just "paid" the debt for the *Accessibility* knowledge , also known as the *Access Modifier* . From the next lesson we will use many of this's accessibility knowledge.

Lesson 24: Final keyword

You probably remember the lesson about *constants* . By then you already know that to declare a constant you must use *final* keyword . Until the OOP lessons, constants are still used intact if you declare a property of the class with this *final* keyword .

So, besides being a constant, what *final* can do in these OOP relationships.
Please watch the lesson this to better understand the *final* .

I. Final Variables & Final Properties

As I said, the keyword *final* when used for a *variable* in a method, has the same meaning as when using this keyword with a *property* of the class. It is both an immutable constant.

Below is an example of the *final* declaration for a *variable* , this ancient knowledge you must have already grasped very well.

```
protected float calculateParimeter() { final float PI = 3.14f;  
return 2 * PI * radius;  
}
```

As for declaring the *final* for an *attribute* , it is equally archaic, you have been used a lot when declaring *PI* in the *Circle* class in the previous lessons.

However, you should remember that *the declarations of the ability to access the final or non-final properties are completely similar* .

```
class Circle {  
private final float PI = 3.14f; protected float radius;  
// Constructor  
public Circle(float radius) { this.radius = radius; }  
protected float calculateParimeter() { return 2 * PI * radius;  
protected float calculateArea() { return PI * radius * radius; }  
}
```

And as I also advise you when talking about *constants* , that you *should capitalize all variable or property characters with final declaration* , it helps our code to be clearer and more coherent.

- *Final Attribute Blank*

In this section I want to talk more about the *final* when used for variables or properties. That is, *when you declare them as final, you do not need to immediately specify a value for that property or variable* , but can be left blank, so that later when there is a specific value, you assign to the property. or the final variable later, and only assign it once. This makes this seemingly very rigid *final* become a bit more flexible.

The following example simulates setting the *final* accurate *PI* value by method. *PI variable* after we find a more

```

protected final float calculateParimeter() { final float PI;
PI = calculatePI();
return 2 * PI * radius;

}

```

The following example simulates setting *the final official PI value* , but slightly different from the above *variable* example , that *for the final property, you are only allowed to assign. following value for final in constructor only* .

```

PI property value after finding the
class Circle {
private final float PI; protected float radius;
// Constructor
public Circle(float radius) { PI = (float) calculatePi(); this.radius = radius; }

protected float calculateParimeter() { return 2 * PI * radius;
}

protected float calculateArea() { return PI * radius * radius;
}

// Don't pay too much attention on below code private double calculatePi() {
return Math.PI;
}
}

```

II. Final Method

It's also easy to deduce. If the *property* , when specified as *final* , it means "peg" the value, can not be changed. Then with the *method* , when you specify the *final* for it, it seems that the method cannot be *overridden* , which is equivalent to the fact that you cannot change *the method* from the subclass.

Back to the *Circle* lop , suppose the *calculateParimeter ()* and *calculateArea () methods* are both declared *final* .

```

class Circle {
private final float PI = 3.14f; protected float radius;
// Constructor

```

```

public Circle(float radius) { this.radius = radius; }

protected final float calculateParimeter() { return 2 * PI * radius;
}

protected final float calculateArea() { return PI * radius * radius;
}

}

```

So in the *Cylinder* class normal for *Cylinder* () methods from *Circle* . that inherits from *Circle* , you can see that it is perfectly

to reuse the calculateParimeter () and calculateArea

```

public class Cylinder extends Circle {
    public float height;

```

// Constructor

```

public Cylinder(float radius, float height) { super(radius);
    this.height = height;
}

```

```

public float calculateTheTich() { return calculateArea() * height;
}
}

```

But like I said, the error only really occurs when you *override* a certain *final* method . For example, if you build more em this in *Cylinder* , the system will report an error.

@Override

```

protected float calculateParimeter() { return super.calculateParimeter(); }

```

And by setting a method to this *final* , as you already know, programmers will avoid letting other classes inherit from certain methods of a class. Doing this still helps to share the method with the inheriting subclasses (as long as we don't declare it *private*), but won't allow any modifications, ensuring the original integrity of the method. Have you seen the magic of OOP yet.

Yes, you can specify *final* for any method. But, you should note that, you *cannot* specify the *final* for a constructor , because the *constructor* is never *overridden* .

III. Final class

The *final effect* on the *class* is similar to her effect on the *method* . That is, you will not be able to inherit from any class declared *final* .

Assuming the *Circle* class has a *final* declaration for the class like this, the *Cylinder* class inheriting from *Circle* we already know will report a "whole child" error .

```
final class Circle {  
// ... }
```

If you publish a class, and only let others use them, don't give anyone inheritance to *override* any method, just set that class to *final* .

We conclude the final lesson gently here. However, the lesson is quite useful in applying a *final* keyword to give some sense of information protection of an object. It will be useful when you want to build Java library packages and publish it to multiple users. Then, it is up to you to allow another class to inherit, or override a certain method of the class.

Lesson 25: Getter and Setter methods

This we will look at how to organize *getter* and *setter methods* in Java. What are these two methods and how do they mean in organizing relationships between objects in OOP? Please read together.

I. What is Getter and Setter?

These are the names for the two categories of methods. These are related to the *data encapsulation* feature in object oriented programming, which in English calls this feature *Encapsulation* . Before that, the definitions of *accessibility* were also a form of *data encapsulation* .

So what is *data encapsulation* ? *Encapsulation data is one of the fundamental properties of inheritance, among other things, like Inheritance you already know, Polymorphism and Abstraction. You will get acquainted later* . The *encapsulation* property helps encapsulate the properties of a class. It makes the properties of the class invisible from other classes.

So what does *getter* , *setter* do with *data encapsulation* ? These the *private* properties of a class either not accessible, readable the *getter* method), or writable (via the *setter* method) from the outer classes. . That is called *data*

encapsulation .

methods make

only (through

The above definition should be clear enough for you to understand if this type of method is correct. If you do not really understand about it, please continue reading the following sections.

II. How To Organize Getter And Setter Practices?

To create encapsulation for properties in a class through *getter* and *setter* , follow these steps.

Defines *accessibility* for properties in a *private* class .

- Construct *getter* or *setter method* for each *private* property . With *setter* you should name the method *setNameOfProperties()* . As for *getter* , you should name the method *getNameOfProperties()* .

The appearance of *getter* and *setter* for *private* properties is optional. You may not need to declare any *getter* or *setter* if you don't want the property to be seen by any class. Or you just need to build a *getter* for a *private* property if you want that property to be viewed only, not repaired. Or just build a *setter* for a *private* property if you want that property to be only externally editable and not viewable.

Now, to make it easy to understand, let's look at the *Point* class example , you see, the *x* and *y* properties in this class are specified *private* , and then it must build more *getter* and *setter methods* in below the property declaration.

```
public class Point {  
    private int x; private int y;  
    // getter of property x public int getX() {  
    return x;  
    }  
    // setter of property x public void setX(int x) { this.x = x;  
    }  
    // getter of property y public int getY() {  
    return y;
```

```
}
```

```
// setter of property y public void setY(int y) { this.y = y;
```

```
}
```

```
}
```

Even for the *Circle* class or other geometry classes that you already know, we can build *getter* and *setter* for them.

```
class Circle { private final float PI = 3.14f; private float radius;
```

```
public float getRadius() { return radius;
```

```
}
```

```
public void setRadius(float radius) { this.radius = radius;
```

```
}
```

```
public float calculatePerimeter() { return 2 * PI * radius;
```

```
}
```

```
public float calculateArea() { return PI * radius * radius;
```

```
}
```

```
}
```

So when calling *getter* and *setter methods* from outside the class will be like. Please pay attention to the following *main ()* function . For this example, I have replaced setting the radius of a circle through the *constructor* , by calling the *setter* of the class.

```
public class MainClass {
```

```
public static void main(String[] args) { // Creat new circle object
```

```
Circle circle = new Circle();
```

```
// set radius of circle by setter circle.setRadius(10);
```

```
// Other methods
```

```
float perimeter = circle.calculatePerimeter();
```

```
float area = circle.calculateArea();
```

```
System.out.println("Perimeter of this circle is: " + perimeter +
```

```
", and area is: " + area) ; } }
```

III. Use Eclipse to declare Getter and Setter

As you can see, the naming of a class attribute's *getter* and *setter* doesn't have a specific rule, they should be *getXXX()* and *setXXX()*.

First you can use the *Point* class , or create any class in Eclipse. Please declare one or more *private* properties for it, as follows.

```
3 public class Point {  
4     private int x;  
5     private int y;  
6  
7 }  
8
```

You can see yellow warning icons on the left bar of the editor, respectively for each *private* property . Since the system sees that we declare *private* for the properties, without using them inside the class, there is obviously a problem, since these values are not visible outside the class.

When you keep clicking on one of these alerts, a small dialog appears, select the *Create getter and setter for 'property_name'....*. With the dialog appearing next to this option, leave it as default and click *OK* to finish.

```
3 public class Point {  
4     private int x;  
5     private int y;  
6  
7 }  
8
```

You can verify it yourself. The result of this option helps you to generate both *getter* and *setter* for the respective property. You just do it in turn for the remaining attributes. And if you want to remove a *getter* or a *setter*, just delete that method after creating it by the above steps.

IV. Getter, Setter And Customizations

By now, you must have understood what *getter* and *setter* are. But you know, these *getter* and *setter methods* are not really used to get hat set input for properties of the class. Because they are methods, it is possible to make use of them to do some manipulation before getting, or setting, values.

My example is a class *Student*, with all kinds of constraints on the import / export of names and ages as follows. You pay attention to test the results returned.

```
public class Student {  
    private String name;  
    private String age;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        if (name == null || name.isEmpty()) {
```

```

// if name hasn't been set (null), or is empty
// set name to "Unknow" this.name = "Unknow";

} else {
this.name = name;
}
}

public String getAge() {
if (Integer.valueOf(age) != -1) { // Age is valid
return age;

} else {
return "Invalid age";
}
}

public void setAge(int age) {
//Check age is valid or not,
// if its valid, set it to age, if not, set age to "-1" if (age > 18) {

this.age = String.valueOf(age);
} else {
this.age = String.valueOf(-1);
}
}

```

With the class declaration as above. Suppose I initialize these student objects in the *main()* function as follows.

```

public class MainClass {

public static void main(String[] args) { // Create student object
Student student1 = new Student(); Student student2 = new Student();

// Set name and age to them
student1.setName(""); // Unknow name student1.setAge(23);
student2.setName("Peter"); student2.setAge(17);

// In thông tin các sinh viên
System.out.println("Student 1 has name: " + student1.getName() + ", age: " +

```

```
student1.getAge()); System.out.println("Student 2 has name: " +  
student2.getName() + ", age: " + student2.getAge()); }  
}
```

Guess what the output to the console will look like. Here are the results.

Student 1 has name: Unknown, age: 23

Student 2 named: Peter, age: Invalid age

So we've just gone through our knowledge of building object-oriented *getter* and *setter methods*. As far as I can tell, the *getter* and *setter* are n't super high at all. Certainly looking at the above examples, you may wonder whether these two methods are important. In fact, there has been a lot of controversy as to whether to build *getter* and *setter* for the properties of the class in Java, or just declare it *public* for all properties. Because in general *getter* and *setter* will make the classes more cumbersome. There are some comments that the application performance also decreases with *getter* and *setter*. But as the lesson states, this is one of the ways of encapsulating class data. With *getter* and *setter*, you will be able to specify whether the property is completely hidden, read-only, or writable, or readable and writable. One way to make your code more explicit.

Lesson 26: Static

It can be said that until this's OOP lesson, more and more *keywords have* been introduced to you one by one. Let me help you review a bit of important keywords and names that we have seen.

- Keyword *extends* You will begin to familiarize yourself with this keyword when expressing *inheritance*.
- Tag *this* - Helps reference to the values within the class.
- *Super keyword* - Helps refer to the values of the nearest superclass.
- Annotation *@Overriding* Shows the method being defined as a method of *veto* or *overriding*.
- Keyword *Object* - Used to refer to the class *Object*, the *superlative* class of all classes in Java.
- Keywords *private / protected / public* - Used when defining *ability to access* class values.
- *Final keyword*- Helps define *constants*, or prevent *overriding* inside OOP.

Getter and setter names- Refers to the encapsulation of data through get / set methods for the *private* property.

So this, we continue to learn more a new keyword, *keyword static* , see what this keyword is, and what it does.

I. Static concept

It can be said that one of Vietnamese meanings in a programming language is the most difficult to understand. “*Tĩnh*” - Translates as *Static* . Knownothing.

Actually, the keyword *static* will be applied when you declare the components of the class as below, I will show it clearly. It has a major effect on memory management. Specifically, *static* members *will belong to the memory management of the class, not to the management of the class instance (or object)* .

Still don't understand.

I invite you to look at the following example, maybe through the example you still do not know the use of the *static* keyword , but let's see how it affects the management of values within the class that you have. mentioned above. Notice the *static* and *non-static components* in the *Point* class .

```
public class Point {  
  
    public static String info; public int x;  
    public int y;  
  
}
```

So I said. The *info* property is *static* , and the *Point* class . The properties *x* , *y* are *not* instances, or objects declared from *Point* . it will be under the management of

static , will be under the control of the
The problem becomes clearer when you use these *static* values of *Point* in another class.

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

```

// These x and y properties are only accessible via the instance point2 of Point
class Point point1 = new Point();
point1.x = 10;
point1.y = 20;

// These x and y properties are only accessible via the instance point2 of Point
class Point point2 = new Point();
point2.x = 5;
point2.y = 6;

// The info property is accessed again through the Point class Point.info = " Save
coordinates of the geometry"; }
}

```

Did you see that the *x* , *y* properties , as well as *any other non-static* value you know about, must be called through an instance of the class, such as *point1* , *point2* in the example. Also attribute *INFORMATION* to be accessed directly through the class *Point*, without any show at all .

At this point, you have understood what *static* value is, right. Sure you have guessed a little meaning of *static* .

Next, let's find out what the keyword *static* is for.

II. Uses Of Static Keywords

As some of the information I have presented above, you can see that, by calling the value of the class through the class name, without having to initialize the object of that class, it makes the values possible. *static* declaration is easier to access.

More specifically, these *static* values can be shared by other classes in the application. It has global implications throughout the application. Just look at the example above, you tried to assign *static* value *info* in *Point* with the command line *Point.info = "Save geometrical coordinates"*; Then somewhere else, in another class, for example, you can take this *Point.info* value , or reassign it a new value. Thus *static* value will be shared, shared, by anywhere, within that class or outside the class (as long as the value is not declared *private*).

But having class values freely accessed and modified like this is a double-edged

sword. If you overdo the keyword *static* , you will inadvertently break the OOP guidelines. OOP was originally born to divide the characteristics and responsibilities of each object, to make it easy to manage. The *static* keyword brings properties that are inherently responsible by some object, and are used in general. Therefore, I recommend that *you be very careful when declaring static for any value of the class* .

So where is *static* keyword actually used? I invite you to read through the section below. In each specific tool of *static* keyword , I will find the most practical examples that the *static* keyword brings in the process of building my application.

III. Static Property Declaration

Perhaps there's not much more to say. *The static property is the property that declared the static keyword* . How they are used then we look at the examples below, you will understand clearly.

1. Example Counting Geometry Declared In A Project

Let's return to examples of types of geometry. For example, we have *Geometry* which is the superclass of two classes *Circle* and *Rectangle* . And the requirement of this's example is, every time we create a geometry object, there is a counter that automatically increments indicating the number of shapes being created in a program. Let's take a look at the *Geometry* class , assuming we are not interested in the other lines of computational code, we just focus on the *count* is set *static* inside this class.

```
public class Geometry {  
    public static int count = 0;  
  
    public Geometry() { count++;  
    }  
  
    // Other lines of code // ...  
}
```

Then, for each *constructor* of the subclass, we call its nearest superclass via the *super () method* . This call will increase this variable *count* in the constructor of the parent class.

```

public class Circle extends Geometry {
// Constructor public Circle() { super();
}
// Caculate perimeter and area methods // ...
}
public class Rectangle extends Geometry {
// Constructor
public Rectangle() { super();
}
// Caculate perimeter and area methods // ...

```

} And then in the *main ()* function , try declaring the objects of *Circle* and *Rectangle* comfortably, and then print the variable *count* to the console.

```
public class MainClass {
```

```

public static void main(String[] args) { // Creat objects
Geometry geometry = new Geometry(); Circle circle1 = new Circle();
Circle circle2 = new Circle();
Rectangle rect = new Rectangle();

System.out.println("Total " + Geometry.count + "shapes in program"); }
```

The output *has 4 pictures in the application* . You see how the variable *count* declared *static* will be shared, right?

2. Example Declare Static Configuration Values For The Application

Have you ever wondered what if you want the application to have shared fixed values, it means configuration values for the application. For example, declare a static path to save this file, or declare a path to this web page, or this API path, For all that static configuration needs, you can group them together in a file and declare *static* properties *associated with final* for it.

Continue with the geometry counting example above. Suppose the requirement of the problem now is that *you must not declare more than 5 pictures* . If the application has more than 5 shapes, instead of the command printing out the number of geometry above, you can print out some message. In this case, the maximum number of 5 shapes in the constraint is treated as a static configuration of the application.

First, we'll need to create a class that's just for configuration. I named this class *Geometry* .

```
public class Geometry {  
    // Declare minium shape number and maximum shapes number public static  
    final int MINIMUM_SHAPE= 0;  
    public static final int MAXIMUM_SHAPES= 5;  
  
    // Other configurations if available // public static final xxx xxx  
    // public static final xxx xxx  
    // ...  
}
```

Back to the *main ()* function , let's try to declare more and more geometrical objects. Following are the constraint checks based on static configuration values obtained from the *Geometry* class .

```
public class MainClass {  
  
    public static void main(String[] args) { // Create instances of classes  
        Geometry geometry1 = new Geometry(); Geometry geometry2 = new  
        Geometry(); Circle circle1 = new Circle();  
        Circle circle2 = new Circle();  
        Rectangle rect1 = new Rectangle(); Rectangle rect2 = new Rectangle();  
  
        if (Geometry.count > Geometry.MAXIMUM_SHAPES) {  
            System.out.println("You have declared over the number of geometry allowed!");  
            System.out.println("The minimum number of geometry is: "  
                + Geometry.MINIMUM_SHAPE);  
            System.out.println("The maximum number of geometry is: "  
                + Geometry.MAXIMUM_SHAPES);  
        } else {  
            System.out.println("Total " + Geometry.count + "shape in program."); }  
    }  
}  
} Now you can run the application to verify it.
```

IV. Static Method Declaration

The static method is used with the same characteristics and purpose as *the static property* above. That is.

- The *static methods* are still under the management of the class, not the class shown.
- A method is set to *static* when they are shared by all objects within the application. Such as methods of checking the correctness of certain values, methods of converting currencies, converting units, methods of storing data to memory, methods of connecting to servers, ...

However, there is a very special note in Java that, *all static methods can only call properties that are declared static*. This is easily verified when a long time ago, in *lesson 8*, I had to declare the variable *static name* when it was outside of *main()*, and because *main()* was declared *static* (*main() function*) it is imperative to declare as *static* function, so that the system can access at any time through the class, rather than declaring the instance of the class containing the *main()* function, so if *main()* wants to use to *name*, *name* must also be *static*.

- *Geometry System Unit Conversion Example*

For this example, suppose our application of geometry is more demanding in terms of input and unit computation. And suppose we accept two input units, *inch* and *centimeter*.

In the short term, the *Geometry* class will have to build some more static configuration values, and have to add static method of unit conversion.

```
public class Geometry {  
  
    //  
    public static final int MINIMUM_SHAPE = 0; public static final int  
    MAXIMUM_SHAPES = 5;  
  
    // Other configuration  
    public static final float PI = 3.14f;  
    public static final float INCH_CM = 2.54f; // 1 inch = 2.54 cm  
    public static final int UNIT_CM = 1; // Mark the application using centimet  
    public static final int UNIT_INC = 2; // Mark the application using inch public  
    static int unit = UNIT_CM; // Mark the application which unit is being used  
  
    // The static method converts centimeters to inches public static float
```

```

ConvertCentimetToInch(float cm) { float inch = cm / INCH_CM; return inch;
}

// The static method converts inches to centimeters public static float
ConvertInchToCentimet (float inch) { float cm = inch * INCH_CM;
return cm;
}
}

```

Then, in any geometry class, the following example I build for the *Circle* class, this class will ask the user what geometry unit the user is using, and will output the corresponding information with the conversion information. the other is based on static configurations and static methods in the *Geometry* class .

```

public class Circle extends Geometry {
protected float radius; private Scanner scanner;

// Constructor
public Circle() {
super();
scanner = new Scanner(System.in); }

public void enterRadius() {
// Enter unit you want to use
System.out.println("What unit you want to use? "); System.out.println("\t1 -
Centimet");
System.out.println("\t2 - inch");
Geometry.unit = scanner.nextInt();

// Sau đó nhập bán kính
System.out.println("Enter Radius of Circle: "); radius = scanner.nextFloat();

}

public void printInfo() { if (Geometry.unit == Geometry.UNIT_CM) {
System.out.println("The circle has radius " + radius + " cm");
System.out.println("Equals" + Geometry.ConvertCentimetToInch(radius)
+ " inch"); } else {
}
}

```

```
System.out.println("The circle has radius " + radius + " inch");
System.out.println("Equals" + Geometry.ConvertInchToCentimet(radius)

+ " cm"); }
}
}
```

Please try to build calls to these methods of *Geometry* to verify.

So we have just gone through the concept and usage of the *static* keyword in Java. You can see that *static* values are really easy to use. And my advice is always, you consider, don't overdo these *static* values . If not for general purposes like the above examples above, then you should not use *static* keyword .

Lesson 27: Overloading

If you remember, we learned about *overriding* . This you learned more about *overloading* . Be careful watch out for your mistake. Overriding, overloading, over, and over...

Just kidding, I help you remember a little bit as follows.

- *Overriding* is the subclass that overrides the method of the parent class.
- *Overloading* is overloaded methods.

We invite you to join us in this's lesson. The lesson will help you understand what the concept of *overloading* is. Besides, it also helps you not to confuse *overriding* and *overloading* .

I. What is Overloading?

This is a pretty good concept in OOP. *Overloading allows a class the ability to define methods with the same name, but different parameters passed* . The method here includes the constructor.

The *overloading* technique increases the usability of methods inside a class. Try looking at the following example.

```
public class Circle extends Geometry {
```

```
// No parameters passed,
```

```

// The application must call the methods to import the radius and unit from the
console public void loginRadius () {
}

// There is one parameter passed as radius,
// the application must call the unit input method from the console public void
loginRadius (float banKinh) {
}

// There are two parameters passed as radius and unit,
// The application just assigns these two values to the corresponding properties
public void loginRadius (float banKinh, int donVi) { }

}

```

As I said above, *overloading* allows you to declare multiple methods in a class with the same name, but with different parameters like the *nhapRadius* methods in the above example.

You have also seen, clearly that *overriding* and *overloading* are confusing in that they all refer to *the names of methods*. But *overriding* requires you to name the method of the subclass to match the method name of the parent class, and the parameters passed must be the same. And *overloading* requires naming methods in a duplicate class (nothing to do with the parent class), and the parameters passed to these methods must be different.

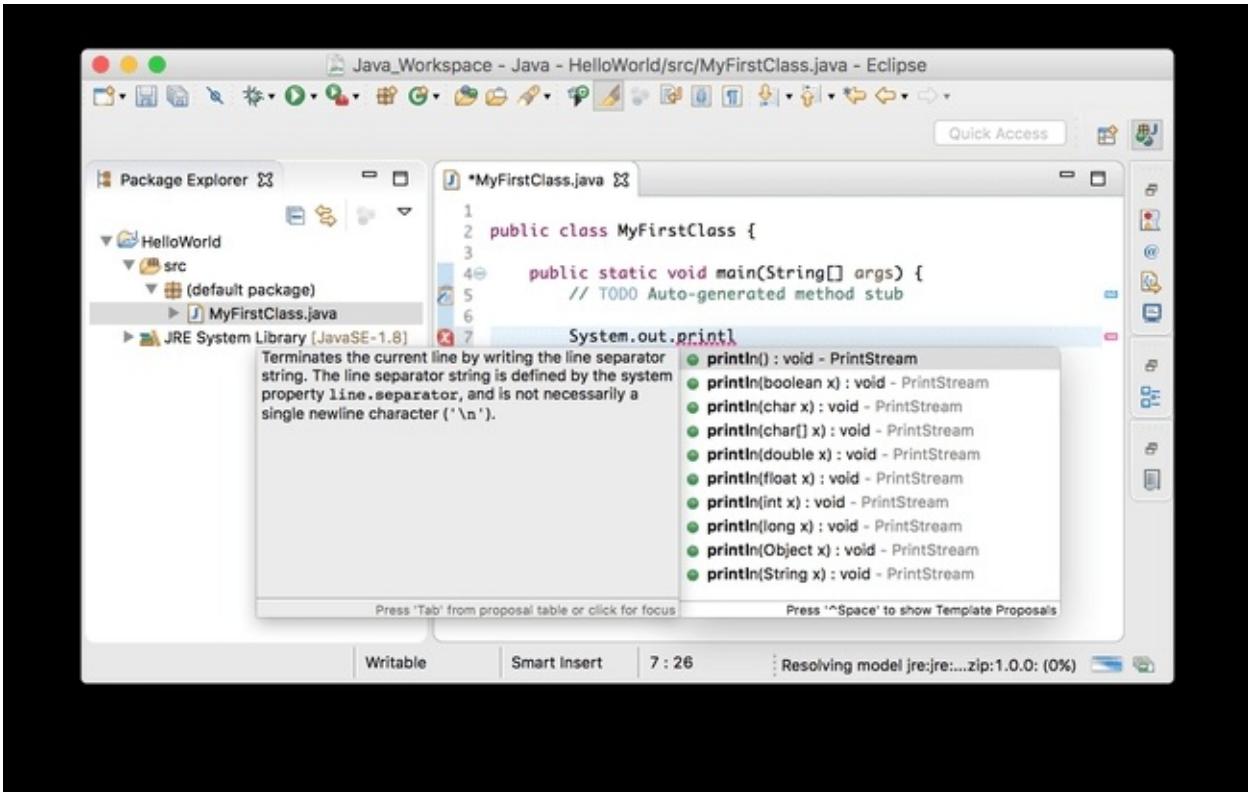
Headache, right. You will get used to it quickly when working with these two techniques a lot.

II. What Does Overloading Methods Do?

As far as I can see. If you apply *overloading* to a class, then build multiple methods of the same name in a class. It will not work immediately for that class, such as it does not make the code of the class tidy or clearer for you to code or manage, sometimes it causes a certain mess. organized within that class, if you have too many methods with the same name.

But *overloading* comes in *handy* when you call them from other classes. It increases the usability of the class using *overloading* technique .

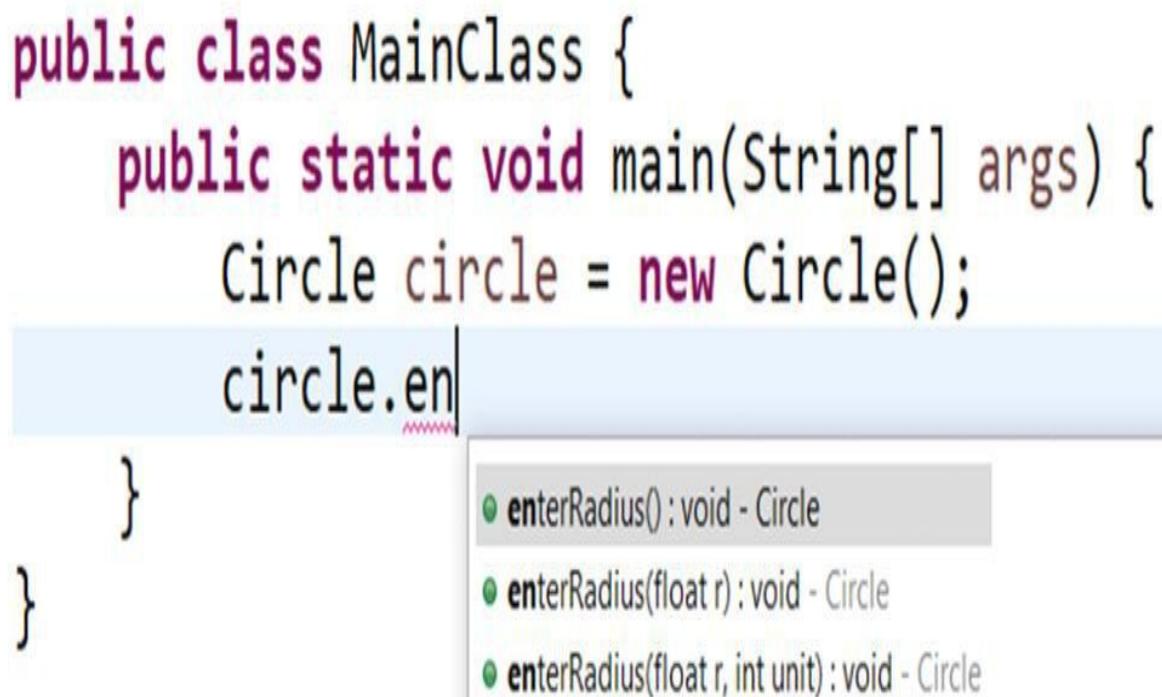
I will give you a demonstration, remember that, every time you call the method to print data to the *console*, you see a lot of options to methods of the same name or not. We invite you to see the suggestions for the *println* method as shown below.



You see, using multiple methods with the same *println* name but with different parameters as above, will increase the efficiency of *System.out*. At this point, the *println* method has "*encapsulated*" all the possible parameters, so you can tell this method to output to the console whatever data you want, it can do without. There is nothing wrong, right.

Go back to the *Circle* class you just tested above, if somewhere else has declared and called the *nhapRadius* methods inside this class, you will see a hint to the three *overloaded* methods as below. It is "*professional*".

```
public class MainClass {  
    public static void main(String[] args) {  
        Circle circle = new Circle();  
        circle.en  
    }  
}
```



A screenshot of an IDE showing Java code. The code defines a class MainClass with a main method that creates a Circle object and calls its enterRadius method. The cursor is at the end of 'circle.en' and a tooltip shows three overloads of the enterRadius method:

- **enterRadius() : void - Circle**
- **enterRadius(float r) : void - Circle**
- **enterRadius(float r, int unit) : void - Circle**

- *Practicing in Building Salary Application for Employees*

It has been a while since we did something new or grand. So this, you should try to move a little muscle. Please open Eclipse and try to build a small project later. Please read the requirements and then try to apply all your knowledge about OOP up until now in practice.

- Program Requirements

Our application serves a small company. With some principles below.

*Note: This example will be present by Vietnamese ‘cause its quite funny

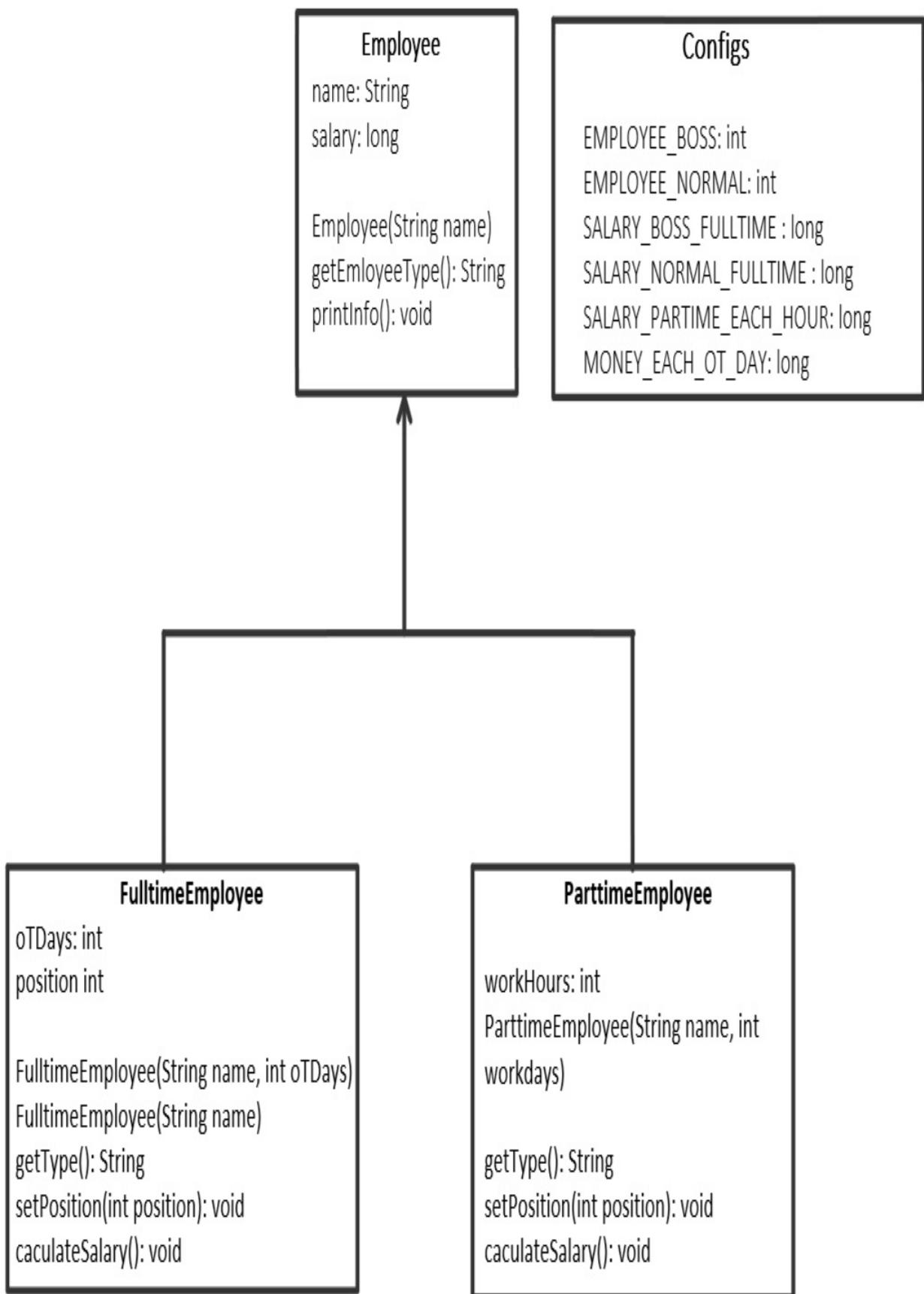
- The company has two types of staff, such as *staff full time* and *staff part time* .

- A *full-time staff* who is a *normal employee* will receive a salary of *10 million* a month. *The full-time staff* who is the *boss* will receive *20 million* a month.
- A *full-time employee* who works overtime any day will be added *800k* per day, regardless of position.
- *Part-time staff* can only work *100k* per hour, regardless of position. Do more, receive more.

That's it, the application will allow entry to each employee. Each employee has the employee's name. There are full-time or part-time staff. A full-time employee is a normal employee or boss employee, are there any extra days? How many hours can a seasonal worker work? Finally based on that information, the corresponding salary screen will be output.

- o Class diagram

Because the program requirements are somewhat complex, it can only be explained most clearly through the *class diagram*.



As you can see, this diagram has a lot more information than it is in your first days. Let me explain some of this additional information.

- *Package* information is shown below class name, with smaller font.

So looking at the diagram we can see that the classes *Employee* , *FulltimeEmployee* and *ParttimeEmployee* are in the same package *model* .

The *Configs* class is in package *util* .

- *Constants* are written in capital value.

Static values will be underlined, like properties in the *Configs* class . So our diagram is clearer and clearer.

o Building Classes

Here you can code it yourself.

The first is the *Configs* class to store static values, such as monthly salary, daily salary, hourly wage, ...

package util;

```
public class Configs {
```

```
// Type of staff
```

```
public static final int EMPLOYEE_BOSS = 1; public static final int  
EMPLOYEE_NORMAL = 2;
```

```
// Staff salary
```

```
public static final long SALARY_BOSS_FULLTIME = 20000000;
```

```
// Boss's monthly salary
```

```
public static final long SALARY_NORMAL_FULLTIME = 10000000;
```

```
// The monthly salary of the staff
```

```
public static final long MONEY_EACH_OT_DAY = 800000; // Working  
overtime for full
```

```
time staff every day is 800k
```

```
public static final long SALARY_PARTIME_EACH_HOUR = 100000; // Part-  
time staff
```

```
salary per hour 100k
```

```
}
```

Next is the *Employee* class . package model;

```

public class Employee {
protected String name; protected long salary;
public Employee () {
}

public Employee (String name) { this.name = name;
}

protected String getEmployeeType () {
// The subclass must override to handle this kind of employee return "";
}

public void printInfo() {
System.out.println ("===== Employee:" + name + "====="); System.out.println
("-Employee's type:" + getEmployeeType()); System.out.println ("- Salary:" +
salary + "VND");

}
}

```

Then there are two subclasses *FulltimeEmployee* and *ParttimeEmployee* .

package model;

import util.Configs;

```

/***
* FulltimeEmployee is a full-time employee */

public class FulltimeEmployee extends Employee {
private int oTDays; // The employee's overtime day private int position; // The
position is normal employee or boss public FulltimeEmployee (String name) {
super (name);
this.position = Configs.EMPLOYEE_NORMAL; // The default is normal
employee }


```

```

public FulltimeEmployee (String name, int oTDays) {
super (name);
this.oTDays = oTDays;
this.position = Configs.EMPLOYEE_NORMAL; // The default is normal
employee

```

```

}

public void setPosition (int position) { this.position = position;
}

@Override
public String getEmployeeType () {
if (position == Configs.EMPLOYEE_NORMAL) {

return " Fulltime Employee " + (oTDays > 0? "with overtime": ""); } else {
return " Fulltime Boss " + (oTDays > 0? "(with overtime)": ""); }
}

public void caculateSalary () {
if (position == Configs.EMPLOYEE_NORMAL) {
salary = Configs.SALARY_NORMAL_FULLTIME + oTDays *
Configs.MONEY_EACH_OT_DAY; } else if (position ==
Configs.EMPLOYEE_BOSS) { salary = Configs.SALARY_BOSS_FULLTIME
+ oTDays * Configs.MONEY_EACH_OT_DAY; }
}
}

package model;

import util.Configs; /**
 * ParttimeEmployee
 */

public class ParttimeEmployee extends Employee {
private int workHours; // Total working hours of staff

public ParttimeEmployee (String name, int workHours) { this.name = name;
this.workHours = workHours;

}

@Override
public String getEmployeeType () { return "Parttime Employee"; }

public void caculateSalary () {
salary = Configs.SALARY_PARTIME_EACH_HOUR * workHours;
}
}

```

```
}
```

And this is the call coming from the *main () method* .
package main;

```
import model.FulltimeEmployee; import model.ParttimeEmployee; import  
util.Configs;
```

```
public class MainClass {
```

```
    public static void main(String[] args) {  
        // The company has 3 full-time employees, including 1 boss,  
        // the boss does not work overtime  
        FulltimeEmployee boss = new FulltimeEmployee("Mr.Boss ");  
        boss.setPosition(Configs.EMPLOYEE_BOSS);  
        FulltimeEmployee normalEmployee1 = new FulltimeEmployee("Mr.Normal  
Employee1");  
        // Mr.Normal Employee doesn't OT
```

```
        FulltimeEmployee normalEmployee2 = new FulltimeEmployee("Ms.Normal  
Employee2", 3);  
        // Ms.Normal Employee2 has 3-days overtime  
        // The company is hiring 1 seasonal employee
```

```
        ParttimeEmployee parttimeEmployee = new ParttimeEmployee("Ms.Parttime",  
240); // Don't question, she is really hard- working
```

```
        // Calculate salary for everyone  
        boss.calculateSalary();  
        normalEmployee1.calculateSalary(); normalEmployee2.calculateSalary();  
        parttimeEmployee.calculateSalary();
```

```
        // Print information of everyone  
        boss.printInfo();  
        normalEmployee1.printInfo(); normalEmployee2.printInfo();  
        parttimeEmployee.printInfo();
```

```
}  
}
```

Here are the results when you execute the program.

```
==== Employee: Mr.Boss ====
-Employee's type: Fulltime Boss
-Salary: 20000000 VND
==== Employee: Mr.Normal Employee1 ====
-Employee's type: Fulltime Employee
-Salary: 10000000 VND
==== Employee: Mr.Normal Employee2 ====
-Employee's type: Fulltime Employee (with overtime)
-Salary: 12400000 VND
==== Employee: Ms.Parttime ====
-Employee's type: Parttime Employee
-Salary: 24000000 VND
```

We just went through another interesting knowledge of Java, knowledge of method *overloading* , or also call *overloading* . Through the lesson, you also know and distinguish well how is *overriding* and what is *overloading* , right?

Lesson 28: Polymorphism

This, we will talk in depth about *Polymorphism* in Java. It seems difficult to hear this feature. Partly because their application is not much. With a name that doesn't sound fixed, like a transformation. Plus quite a few documents have been written about this function of OOP.

Then let's dive into the lesson to see what *Polymorphism* is and is it really difficult.

I. What Is Polymorphism?

This time, the English and Vietnamese meanings in Java programming match. Not many words rambling, only *Polymorphism*, or *Polymorphism* only.

So why *polymorphism*? As you know, OOP is inherently a way of real-oriented programming thinking, so obviously its concepts must also be close to the features in practice. Including *Polymorphism*. In fact, the *Polymorphism* is seen as a special object, at times this object takes some form (becomes an object), and at times this object takes another form, depending on the circumstances. These "*immersion*" in different shapes (objects) help the *Polymorphic* objectInitially it is possible to perform different actions of each specific object. For example, if in your company, there are employees who accept two different responsibilities, they are both full-time employees on weekdays, but work part-time on weekends. So, to calculate the salary for this employee, depending on each time the system will consider the employee as full-time or part-time, and the salary calculation method of each employee will perform the calculation. The best results depend on each of these different roles. You also have a rough understanding of *Polymorphism*, right.

One thing is for sure. If we do not consider the salary calculation action of employees as the above example is *Polymorphism*, we can still build a complete payroll system, but it will be more complicated than if you know *polymorphism*. what.

And one more idea. That the *Polymorphism* of this's lesson is also one of the outstanding features that OOP brings. You try to capture and take advantage of. A quick review of OOP's core features include:

- *Encapsulation* calculation. This property is demonstrated through the knowledge of *accessibility*, *getter / setter*.
- Calculate *Inheritance (Inheritance)*. This property is demonstrated through the knowledge of *inheritance*, *overriding*, *overloading*.
- *Polymorphism*. This we will learn.
- *Abstraction*. Next lesson we will learn.

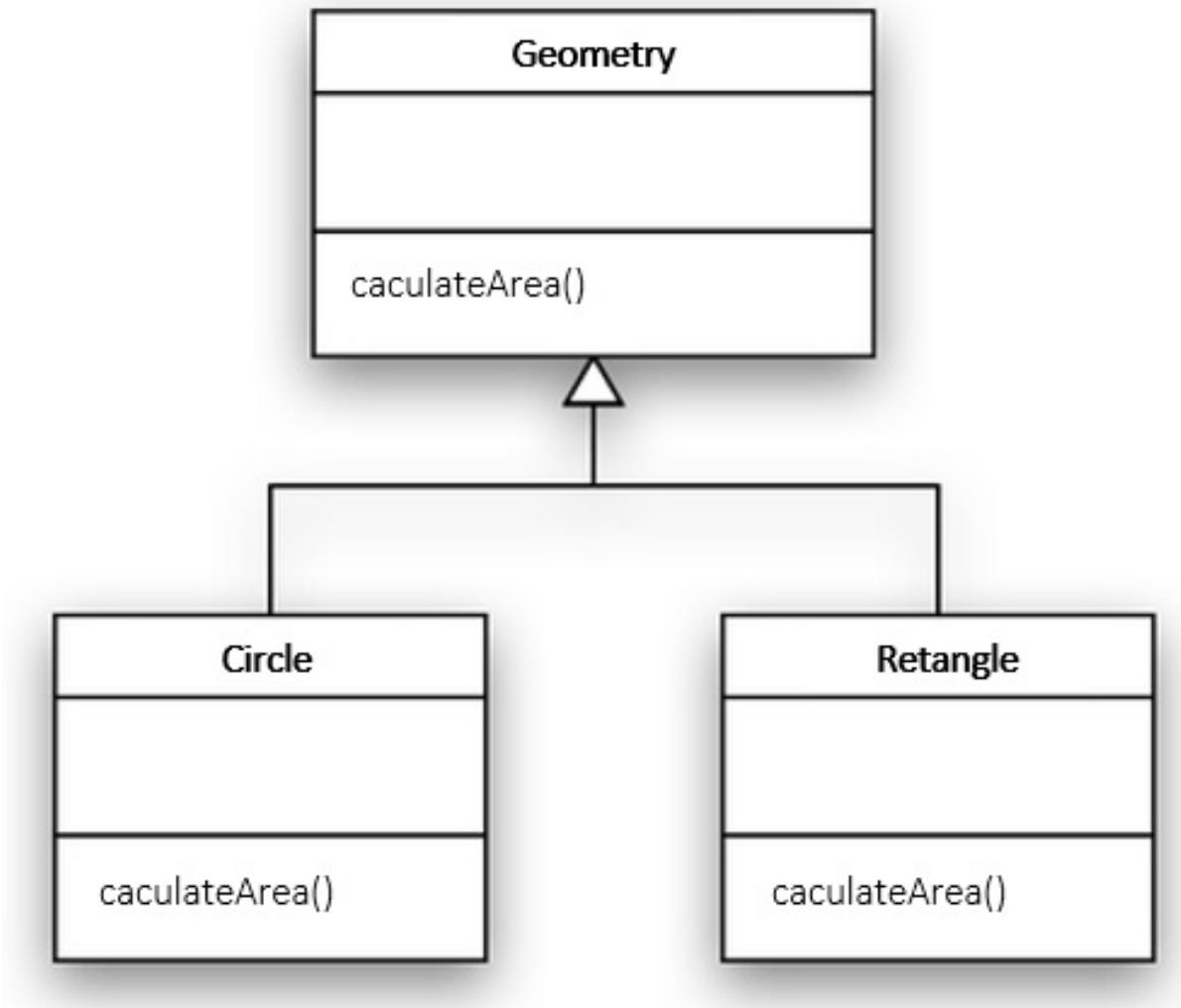
II. How to Use Polymorphism?

By now, you should have a rough understanding of the concept of *Polymorphism*. So in OOP how do we organize and use this *Polymorphic* property?

First, *Polymorphism* will be associated with *inheritance*. And, *Polymorphism* also attached to *overwrite method* (overriding) again. Because as mentioned above, *Polymorphism* is referring to a certain object capable of becoming other objects. So in order for an object to be a certain object, it must be a parent object. And for the parent object to be one of the child objects in each situation, it must define methods for its children to override. This helps the system determine which objects and methods are actually running while the application is running. So many documents called *Polymorphism* is *Polymorphism at runtime* is.

We will come to the following example to make it easier to understand. The example is quite simple.

The *Geometry* class is the parent class, the two subclasses *Circle* and *Rectangle* both override the *caculateArea()* method from the parent.



Their code is quite simple too, we remove all other mustaches, focusing on override methods only.

- *Geometry*

```
public class Geometry {  
    public void caculateArea() {  
        System.out.println("Unknow shape");  
    }  
}
```

- *Circle*

```
public class Circle extends Geometry {  
    @Override  
    public void caculateArea() {
```

```

System.out.println("This is area of Circle");
}
}
- Rectangle
public class Rectangle extends Geometry {
@Override
public void caculateArea() {
System.out.println("This is area of Rectangle ");
}

```

Now, the magic of *Polymorphism* is here, please pay attention to the code that declared and use the overriding methods above as follows.

- *MainClass*

```

public class MainClass {
public static void main(String[] args) { Geometry geometry = new Geometry();
// Print "Unknow Area" geometry.caculateArea();

// Sometime, geometry become another type of shape, this time is a Circle
geometry = new Circle();
geometry.caculateArea(); // This will print "This is area of Circle"

```

```

// Sometime, geometry become another type of shape, this time is a Rectangle
geometry = new Rectangle();
geometry.caculateArea(); // This will print "This is area of Rectangle"

```

}

} You see, the *Geometry* object itself has a method *caculateArea ()* . But unlike using objects from lessons so far, that when we need subclasses to do the area calculation, we will declare the subclass and call the overridden method on the subclass. Then in this's lesson we allow the *Geometry* class to act as a subclass, by re-initializing the object as its subclass, *Geometry geometry = new Circle ()* , then itself will act as subclass there. Count *Polymorphism* is here.

- *Practicing in Building Salary Application for Employees*

If in *the previous lesson* we have built a "*complete*" payroll system "*complicated*" for a "*big*" company .

But the code at that time was not very practical, because we coded "*hard*" by knowing in advance which employees are employees, who are bosses, who work

full-time, who work for sale. time, to which the corresponding *FulltimeEmployee* or *ParttimeEmployee* objects are declared .

So in this's lesson, we will complete the employee salary calculation application from the previous lesson. Make the system more realistic. Specifically, in this lesson we will let the user enter employee information manually. And so there will be an array of employees in the app. The *Employee* class will be the class that uses the *Polymorphic* property to be able to act as *FulltimeEmployee* or *ParttimeEmployee* in each particular situation.

o Describe Program Requirements

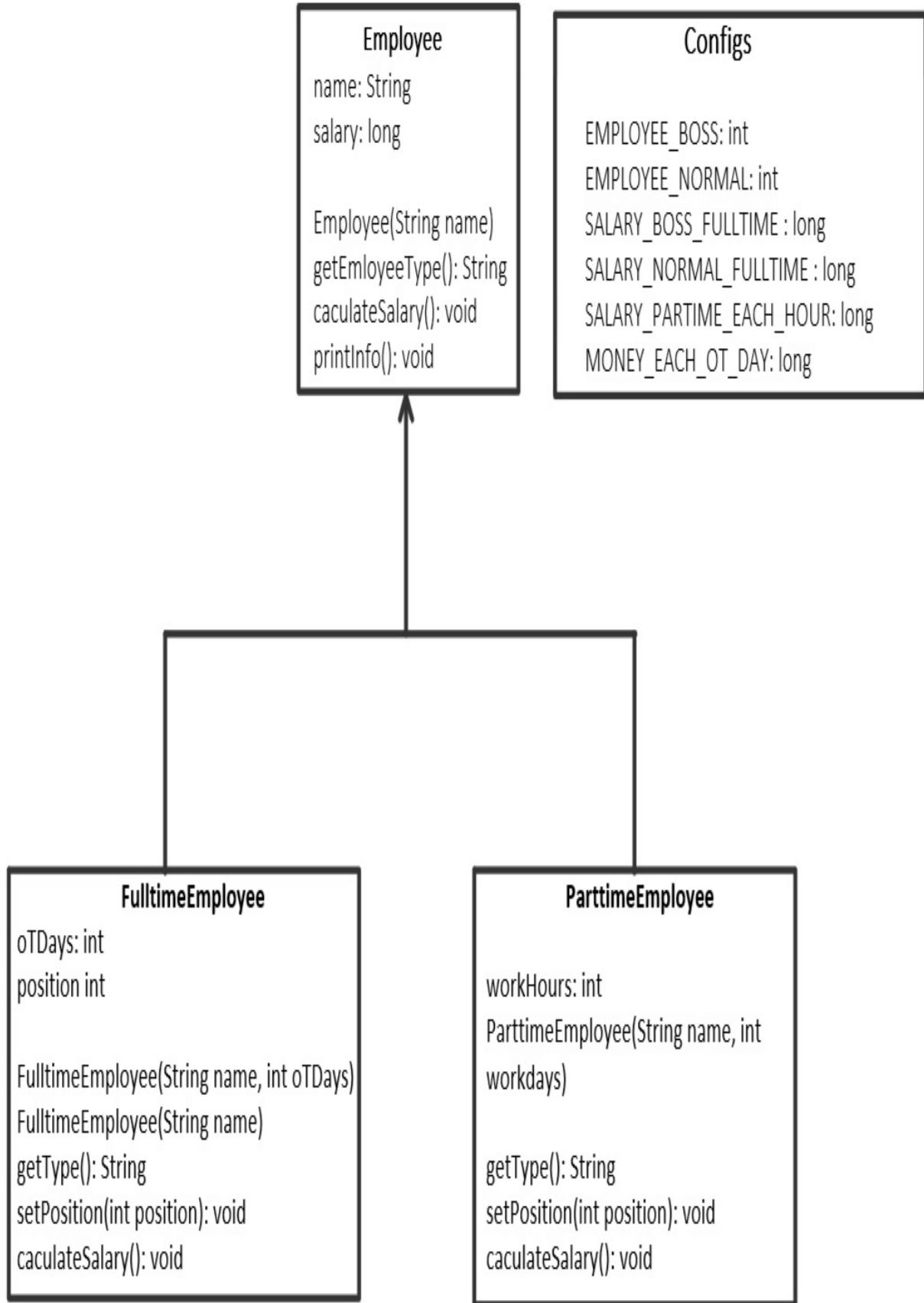
The requirements of the payroll program have not changed from the previous lesson. I just described it.

- The Company has two types of employees: *employees full time* and *employee time* .
- A *full-time employee* who is a *employee* will receive *10M* per month. *The full-time employee*, who is the *boss* will receive *20M* per month.
- A *full-time employee*, who works overtime any day will receive *800k* per day, regardless of position.
- *Parttime employee* can only work *100k* per hour , regardless of position. Do more, enjoy more.

Application will allow the user to enter the number of employees. Then for each employee, the user must enter the employee's name, the type of employee full-time or part-time, the full-time employee is the employee or boss employee, do you have any overtime work? how many hours can you do seasonal workers? Finally based on that information, the corresponding salary screen for all employees will be displayed.

o Class diagram

We still rely on the class diagram from the previous lesson. But modify it a bit so that the *Employee* class will "play" well in its subclasses. By constructing the *calculateSalary () method* in this class, then in the subclasses will have to override again.



- o Building Classes

The *Configs* class hasn't changed.

```
package util;
```

```
public class Configs {
```

```
// Type of employee
```

```
public static final int EMPLOYEE_BOSS = 1; public static final int  
EMPLOYEE_NORMAL = 2;
```

```
// Employee salary
```

```
public static final long SALARY_BOSS_FULLTIME = 20000000;
```

```
// Boss's monthly salary
```

```
public static final long SALARY_NORMAL_FULLTIME = 10000000;
```

```
// The monthly salary of the employee
```

```
public static final long MONEY_EACH_OT_DAY = 800000; // Working  
overtime for full
```

```
time employee every day is 800k
```

```
public static final long SALARY_PARTIME_EACH_HOUR = 100000; // Part-  
time
```

```
employee salary per hour 100k
```

```
}
```

Class *Employee* only add methods *caculateSalary()* to perform *Polymorphism* on this method.

```
package model;
```

```
public class Employee {
```

```
protected String name; protected long salary;
```

```
public Employee () {
```

```
}
```

```
public Employee (String name) { this.name = name;  
}
```

```
public void caculateSalary() {
```

```
// Sub-class must override this method
```

```
}
```

```
protected String getEmployeeType () {
```

```
// The subclass must override to handle this kind of employee return "";
```

```

}
public void printInfo() {
    System.out.println ("===== Employee:" + name + "====="); System.out.println
    (" -Employee's type:" + getEmployeeType()); System.out.println (" - Salary:" +
    salary + "VND");
}
}

```

The *FulltimeEmployee* and *ParttimeEmployee* classes have not changed either.

Only reduce *overloading* in *constructor* to make input easier.

FulltimeEmployee and *ParttimeEmployee* classes :

```

package model;
import util.Configs;

```

```

/ **
* FulltimeEmployee is a full-time employee * /

```

```

public class FulltimeEmployee extends Employee {
    private int oTDays; // The employee's overtime day private int position; // The
    position is normal employee or bos

```

```

public FulltimeEmployee (String name, int oTDays) {
    super (name);
    this.oTDays = oTDays;
    this.position = Configs.EMPLOYEE_NORMAL; // The default is normal
    employee
}

```

```

public void setPosition (int position) { this.position = position;
}

```

@Override

```

public String getEmployeeType () {
    if (position == Configs.EMPLOYEE_NORMAL) {
        return " Fulltime Employee " + (oTDays > 0? "with overtime": "");
    } else {
        return " Fulltime Boss " + (oTDays > 0? "(with overtime)": "");
    }
}

```

```

}

public void caculateSalary () {
if (position == Configs.EMPLOYEE_NORMAL) {
salary = Configs.SALARY_NORMAL_FULLTIME + oTDays *
Configs.MONEY_EACH_OT_DAY; } else if (position ==
Configs.EMPLOYEE_BOSS) { salary = Configs.SALARY_BOSS_FULLTIME
+ oTDays * Configs.MONEY_EACH_OT_DAY; }
}
}

package model;
import util.Configs;

/**
 * ParttimeEmployee
 */
public class ParttimeEmployee extends Employee {
private int workHours; // Total working hours of employee

public ParttimeEmployee (String name, int workHours) { this.name = name;
this.workHours = workHours;

}

@Override
public String getEmployeeType () { return "Parttime Employee"; }

public void caculateSalary () {
salary = Configs.SALARY_PARTIME_EACH_HOUR * workHours;
}
}

```

And here. Any changes will be made to the *main () method* . If you do not pay attention to the other code entered from the console. The *main () method* has the following ideas we should pay attention to.

For the first time ever, we use an *array of objects* . And you can see that the array of objects is no different *from the array of primitive types* that you have learned.

- For each element in the *Employee* array .

We initialize this *Employee* as *FulltimeEmployee* or *ParttimeEmployee* due to the condition that the user enters. Count *Polymorphism* effective here.

```
package main;
import java.util.Scanner;

import model.Employee;
import model.FulltimeEmployee; import model.ParttimeEmployee;

public class MainClass {

    public static void main (String [] args) {
        // Ask user to enter number of employee
        Scanner scanner = new Scanner (System.in);
        System.out.print ("Please enter the number of employees:"); int
        numberEmployees = Integer.parseInt (scanner.nextLine ());

        // Declare an array of employee
        Employee [] arrayEmployees = new Employee [numberEmployees];
        for (int i = 0; i < numberEmployees; i++) {

            // Declare each type of employee, and ask the user to enter employee information
            System.out.print ("Employee name" + (i + 1) + ":");

            String name = scanner.nextLine ();
            System.out.print ("Are employees (1-Full-time; 2-Part-time):");
            int isEmployee = Integer.parseInt (scanner.nextLine ());
            if (isEmployee == 1) {

                // Full time employee
                System.out.print ("Employee position (1-Boss; 2-Normal Employee):"); int
                position = Integer.parseInt (scanner.nextLine ());
                System.out.print ("Overtime (if any):");
                int oTDays = Integer.parseInt (scanner.nextLine ());
                arrayEmployees [i] = new FulltimeEmployee (name, oTDays, position);

            } else {
                System.out.print ("Working hours:");
                int workHours = Integer.parseInt (scanner.nextLine ());
                arrayEmployees [i] = new ParttimeEmployee (name, workHours);
            }
        }
    }
}
```

```
}

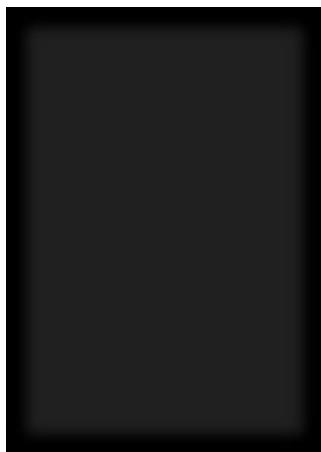
System.out.println ("\nSalary result \ n");

// Caculate employee 'salary and print employee information for (Employee
employee: arrayEmployees) {
employee.caculateSalary ();
employee.printInfo();
}

}

}
```

Finally, the results of program execution. In addition to the dynamic input, the output is the same as the previous day. Try combining the dynamic input of this's lesson with not using *Polymorphism* , but using it as the lesson from the previous day. With this test, you will better understand the strengths of *Polymorphism* .



Polymorphism is that, do you find it difficult. There will also be knowledge related to *Polymorphism* , such as casting in OOP, which we will talk about in the next lesson.

Lesson 29: Casting (Again) in OOP

As you remember, we talked about *casting* in *lesson 6* . And obviously you already know what the *concept of casting* , we don't need to mention this lesson.

And casting in that lesson is casting on *primitive data* . Casting at that time was distinguished into two separate cases: *implicit casting* and *explicit casting* .

Apparently, the casting you already know isn't that complicated either. So this, when we learn about OOP, we will see what casting on *non-primitive data* looks like.

I. Implicit Castring

Well, in OOP, we can also classify casting into two types, *explicit* and *implicit*.

Implicit casting with classes in OOP similar *implicit casting* with *primitive data*. That is, if there is no data loss, or can exnamed the data storage capacity, then it seems that the system will completely *implicitly* help us to cast.

However, with OOP, we have the following two *default* casting cases, inviting you to go to the two sub-sections below.

1. *Implicit casting among same class object*

Actually this issue is also familiar. It's like you declare two variables *int a* and *int b*, and then you assign *b = a*, and the result is *a* and *b* will have the same value. With OOP, too, it will be like you declare *Employee a* and *Employee b*, then you assign *b = a*, the result is *b* with *a* will bring the same connamet.

However, but not really. It is normal to assign two primitive variables *int to int* as above. But the fact that you assign two *Employee* objects to *Employee* is considered as a casting, so let's try the example below.

Suppose there is an extremely one *name* property and *getter / setter* the *name* information to the console. simple *Employee* class, which has only

methods for it, and a method that outputs

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

```
public void setName(String name) { this.name = name;  
}
```

```
public void printInfo() {  
System.out.println("Employee: " + name);  
}
```

Go to *main()* method.

We will declare two objects *employee1* and *employee2* grades *employee* this, and you see the system assigns or implicit casting two objects for each other with lines of code later.

```
public class MainClass {  
  
    public static void main(String[] args) {  
        // Declare 2 object employee1 and employee2 from Employee Employee  
        Employee employee1 = new Employee();  
        Employee employee2 = new Employee();  
  
        // Set name for them  
        employee1.setName("David"); employee2.setName("Jane");  
  
        // Print output information of two objects first time: // can you guess the correct  
        // output?  
        employee1.printInfo(); // "Employee: David" employee2.printInfo(); //  
        // "Employee: Jane"  
  
        // Assign two employee objects, the system will also force // the default type  
        // employee1 to employee2  
        employee2 = employee1;  
  
        // Export the information of two objects for the second time: // the result is not  
        // strange, because employee2 will contain the same content // as employee1  
        employee1.printInfo(); // "Employee: David"  
        employee2.printInfo(); // "Employee: David"  
  
        // Change the name value information of employee2  
        employee2.setName("Sean");  
  
        // Information output for two objects third times:  
        // See how weird it is? both objects have their property values changed  
        employee1.printInfo(); // "Employee: Sean"  
        employee2.printInfo(); // "Employee: Sean"  
    }  
}
```

You pay atnametion to the comment "*output information of two objects 3 times*" . Before information at this time

every *employee2* is to change
for both subjects *employee1* and *employee2* , only

the value to attribute *annamena* only, so excuse the how *name* in *employee1* was fate ? Since you should remember that, in OOP, when you assign two objects to each other, such as the assignment *employee2 = employee1* , *the system coerces the data type, and also the reference, of both objects. together, making them one* . This is completely different from using the in assignment *primitive data type* , please keep this in mind.

2. Squeeze The Class From Child Class To Father's Class

Like format *implicit* casting expands the storage capacity of *primitive data types* . The system will perform casting when there is a conversion of data from small data types to larger data types, such as from *int* to *float* . Then the system will do the same with OOP, if there is data conversion from subclass to superclass.

Now suppose we modify the *Employee* subclass, which is the *FullTimeEmployee* class . So that this subclass *overrides* the *printInfo () method* of the parent class. The code of this *FullTimeEmployee* class is as follows.

```
public class FullTimeEmployee exnameds Employee {  
    @Override  
    public void printInfo() { System.out.println("Full-time employee: " + name); }  
}
```

Already. So we have a look at the *main () method* , see how *the implicit* casting takes place. You can see that the outcome of the assignment, and casting between classes in OOP both results in two objects becoming one (because of the same reference). After an assignment, whenever you change the value of one class, the same class in the other assignment will have the same value changed.

```
public class MainClass {  
  
    public static void main(String[] args) {  
        // Declare 2 employee  
        Employee employee = new Employee();  
        FullTimeEmployee employeeFullTime = new FullTimeEmployee();  
  
        // Set name for them
```

```

employee.setName("David"); employeeFullTime.setName("Jane");

// print information first time
employee.printInfo(); // "Employee: David" employeeFullTime.printInfo(); //
"FullTime Employee: Jane"

// Cast the default type from FullTimeEmployee to Employee, completely
automatically employee = employeeFullTime;

// print information second time
employee.printInfo(); // "FullTime Employee: Jane"
employeeFullTime.printInfo(); // "FullTime Employee: Jane"

// Change the name value within employee2 employee.setName("Sean");

// print information third time
employee.printInfo(); // "FullTime Employee: Sean"
employeeFullTime.printInfo(); // "FullTime Employee: Sean"

```

II. Explicit Casting

You can totally guess when we need *explicit* casting, right. That is when the system detects that you are trying to convert data from a larger data type to a smaller one. With OOP, it is from parent class to subclass.

We come to the example with the opposite assignment to the example just above.

```

public class MainClass {

    public static void main(String[] args) {
        // Declare 2 employee
        Employee employee = new Employee();
        FullTimeEmployee employeeFullTime = new FullTimeEmployee();

        // set name for them
        employee.setName("David");
        employeeFullTime.setName("Jane");

        // print information first time
        employee.printInfo(); // "Employee: David" employeeFullTime.printInfo(); //
        "FullTime Employee: Jane"
    }
}

```

```
// Explicit casting from Employee to FullTimeEmployee employeeFullTime =  
(FullTimeEmployee) employee; }  
}
```

Why is the code so little this time. Actually you should not code anymore, an error occurred on the last line. Even after you code, the compiler doesn't give any error, but if you execute the application right now, you will get an error like this.

Employee: David

Fulltime Employee: Jane

Exception in thread “main” java.lang.ClassCastException: model.Employee cannot be cast to model.FullTimeEmployee at main.MainClass.main(MainClass, java: 25)

This error means that when you explicitly *cast the word employee* to *employeeFullTime* , the compiler still makes sense now, because they are parent-child. But when executed in

This error means that when you explicitly cast the word *nhanVien* to *nhanVienFullTime* , the compiler still makes sense now, because they are parent-child. But when executed in the real environment, the *nhanVien* class inherently doesn't know what *nhanVienFullTime* is, so you can't perform casting .

So what is *explicit* casting for OOP? In fact, if we apply *polymorphism* , that is, at some point *employee* has to "put himself" in the role of a *employeeFullTime* , then they will understand each other when "together walking in the life path" ahead. . Code like this should work fine.

```
public class MainClass {  
  
    public static void main(String[] args) {  
        // Declare 2 objects  
        Employee employee = new Employee();  
        FullTimeEmployee employeeFullTime = new FullTimeEmployee();  
  
        // Set name for them  
        employee.setName("David"); employeeFullTime.setName("Jane");  
  
        // Print infomation of 2 object lần 1  
        employee.printInfo(); // "Employee: David" employeeFullTime.printInfo(); //
```

"FullTime Employee: Jane"

```
// Explicit casting from Employee to FullTimeEmployee, // but Employee must  
be polymorphic before employee = new FullTimeEmployee();  
employeeFullTime = (FullTimeEmployee) employee;
```

```
// Change the name value of employee employee.setName("Sean");
```

```
// Print infomation of 2 object lần 2
```

```
employee.printInfo(); // "FullTime Employee: Sean"
```

employeeFullTime.printInfo(); // "FullTime Employee: Sean" This's lesson is just like that. Also relatively complicated, right. However, do not worry too much, the appearance of casting in OOP is not much. Your job now is to read and understand the constraints on this's lesson. Then later on when you find yourself in a particular situation, or if you look at the source code somewhere that demonstrates the coercion in OOP like this, then you will catch the problem.

Lesson 30: Abstraction in Java

By going through this's lesson, we will also cover *four core features of object oriented programming* . The four characteristics are:

- Calculate the *packed data* , also called *Encapsulation* . You can review this property in lessons: *accessibility* , *getter / setter* .
- Calculate *Inheritance* , also called *Inheritance* . You can review this property in the following lessons: *inheritance* , *overriding* , *overloading* .
- Calculate *Polymorphism* , aka *Polymorphism* . You can review this property in the lesson on *polymorphism* .
- And this. Count *Abstract* , aka *Abstraction* .

Upon hearing it, it is very abstract. Is this knowledge difficult to understand and does it help to organize code in OOP? Let's go to the lesson together.

I. What Is Abstraction?

In fact *abstraction* can also be understood as something which is *not real* . So the *Abstraction* in OOP means to refer to a certain class with an abstract property, not real. And so you can understand that the *abstract* class will be a class that doesn't exist right?

Actually, the *Abstract* class still exists, it is still a class. But it's abstract in that it

cannot be used to create objects like any other normal class . The Abstract class is then just a "soulless body" , or you can understand it as just a rib, so that you can create subclasses of it based on the constraints from this rib .

Heard enough to see this abstract already. Let's take a look at what it means to declare a class as *Abstraction* , and what the characteristics of an *Abstract* class will look like in the next item.

II. Abstract Class Declaration Like?

Here is the syntax for you to declare an *Abstract* class . abstract class { properties; methods;

}

You can see, to declare a class as *Abstract* , just prepend the *class* keyword with an *abstract* keyword .

But then what is the purpose of this *Abstraction* , let's go to the following section.

III. Why Be Abstract?

The prerequisite reason that you must know about *Abstraction* is because there are more than 90% of your ability to apply for a job, the interviewer will ask you this question: "*Please help distinguish between the abstract class and the interface*" !!! Just kidding! But they really ask!

Actually, the reason to declare an *Abstract* class is because there is at least one *abstract* method in the class . It can be said on the contrary for easy understanding as follows, *if a class has at least one method defined as Abstract, then that class must declare Abstract* . Here you will have a lot of questions, I would like to answer slowly.

What is Method of Abstraction? It is the method that defines the *abstract* keyword when declared. The syntax for declaring an *Abstract* method is like declaring an *Abstract* class.

[access_link] abstract type of method name ([transfer_address ""]);

What does Abstract Method do? Methods that declare *abstract* will require no function body. As you can see in the syntax above, there will be no information

about the block of this method. You can compare with the syntax of the normal method *here*. You can only declare the name, the parameters (if any), the return type, and then end with (;) for the *Abstract* methods. The binding of *Abstraction* can be seen clearly starting right here. The reason *abstract* methods don't have a functional body, because they don't have to. Once a class inherits from this *Abstract* class, that class must *really* (English called *implements*) content for *all* modes of *abstraction* of this, by *the override* them.

By now you can see that the *Abstract* layer is really not so superhuman. It is just a class that cannot instantiate (cannot instantiate an object from it). But it binds its subclasses to enforce the content for the *Abstract* methods within it. That's all. You can come across this *Abstraction* somewhere when using packages from the system (sometimes called the Java platform) or from other distributors. When you inherit their classes, suddenly the system crashes and forces you to immediately *override some* method of that class, you will surely immediately understand that the class is the *Abstract* class. After a while, we will examine which class is the *Abstract* class in the system, after going through the exercise below.

1. Practicing in Building Salary Application for Employees

Let's go back to the math problem for employees (present by Vietnamese), which in *lesson 30* we have made it whole. In a sense, you don't need to add anything to this exercise. This section is only intended to show you how an *Abstract* class will be declared and used. You do not have to mechanically apply the *Abstraction* as you would in the exercise.

a. Describe Program Requirements

The requirements of the payroll program have not changed from the previous lesson. I just described it.

- The Company has two types of employees: *employees full time* and *employee seasonal (part-time)*.
- A *full-time employee* who is a *solder* will receive a salary of *10* a month. The *full-time employee* who is the *boss* will receive *20* employees a month.
- A *full-time employee* who works overtime any day will be added *800k* per day, regardless of position.
- *Seasonal employee* can only work *100k* per hour, regardless of position. Do more, enjoy more.

Application will allow the user to enter the number of employees. Then for each employee, the user must enter the employee's name, the type of employee full-time or part-time, the full-time employee is the soldier or boss employee, do you have any overtime work? how many hours can you do seasonal workers? Finally based on that information, the corresponding salary screen for all employees will be displayed.

b. App Enhancing By Building Abstract Employee Class

Here is the code of the *Employee* class from the exercise *that day* . package model;

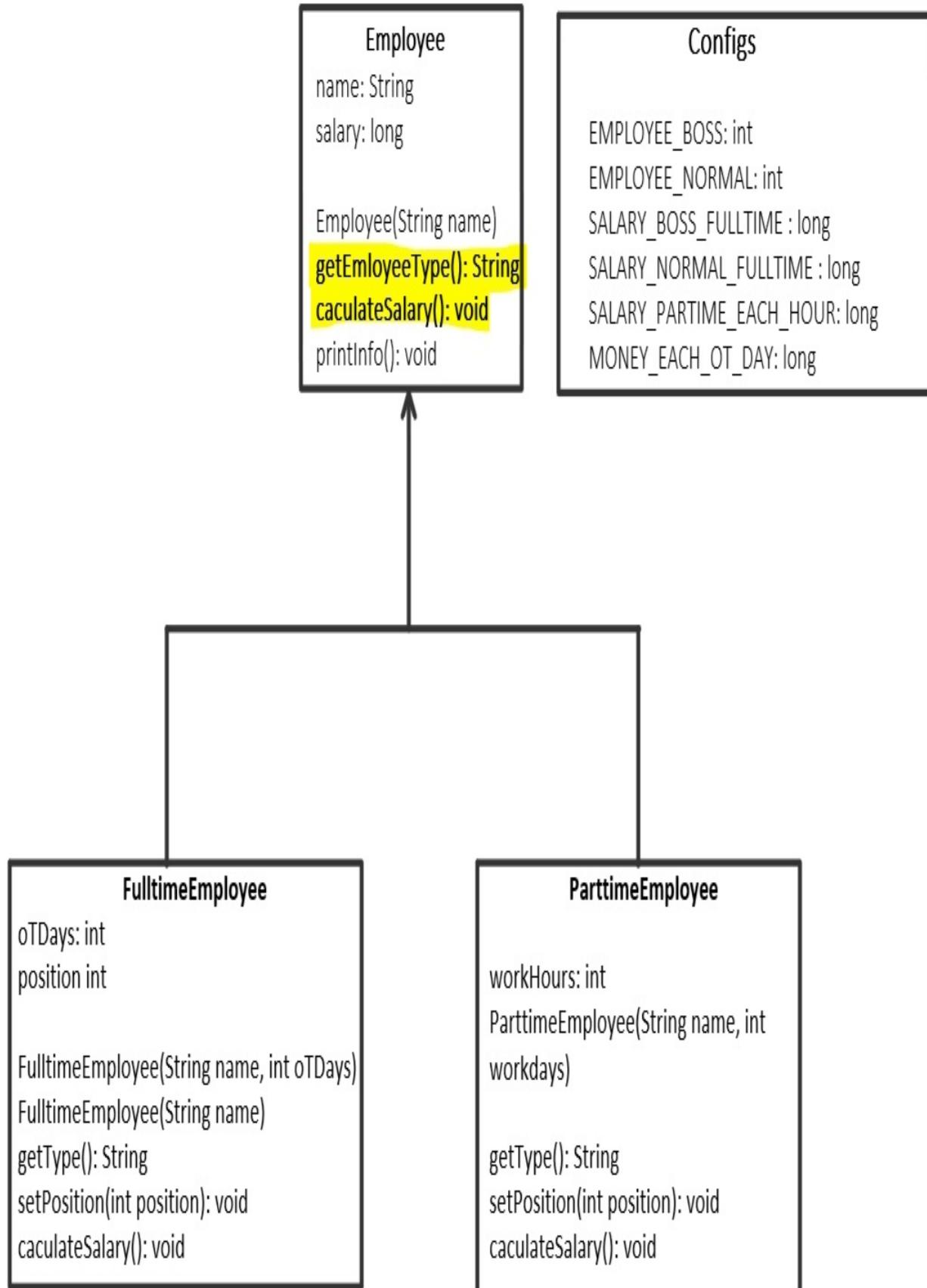
```
public class Employee {  
    protected String name; protected long salary;  
    public Employee () {  
    }  
  
    public Employee (String name) { this.name = name;  
    }  
  
    public void caculateSalary() {  
        // Sub-class must override this method  
    }  
    protected String getEmployeeType () {  
        // The subclass must override to handle this kind of employee return "";  
    }  
    public void printInfo() {  
        System.out.println ("===== Employee:" + name + "====="); System.out.println  
        (" -Employee's type:" + getEmployeeType()); System.out.println (" - Salary:" +  
        salary + "VND");  
    }  
}
```

You see, in the two methods *getEmployeeType()* and *caculateSalary()* circled by me, do you really need subclasses to *override* these two methods? With code like this, there is no constraint, and if someone does the coding of *Employee*'s subclasses , will they remember to *override* these two methods?

So the requirement of this's lesson is clear, we will upgrade this payroll application by declaring *Employee* as the *Abstract* class .

c. Class diagram We still rely on the class diagram from the previous lesson. But the *getEmployeeType ()* and *caculateSalary ()* methods the *Employee* class must also the *Abstraction* declarations will be italicized (I also bold them for easy visibility). would be the *Abstraction* methods . And be the *Abstract* class . In the class of course

diagram,



d. Building Classes

The *Configs* class has n't changed.

package util;

```
public class Configs {
```

```
// Type of employee
```

```
public static final int EMPLOYEE_BOSS = 1;
```

```
public static final int EMPLOYEE_NORMAL = 2;
```

```
// Employee salary
```

```
public static final long SALARY_BOSS_FULLTIME = 20000000;
```

```
// Boss's monthly salary
```

```
public static final long SALARY_NORMAL_FULLTIME = 10000000;
```

```
// The monthly salary of the employee
```

```
public static final long MONEY_EACH_OT_DAY = 800000; // Working  
overtime for full
```

```
time employee every day is 800k
```

```
public static final long SALARY_PARTIME_EACH_HOUR = 100000; // Part-  
time
```

```
employee salary per hour 100k
```

```
}
```

The *Employee* class is then the *Abstract* class . And here is the code for *Employee* class , you can compare with the code image above of this class to see the change.

package model;

```
public abstract class Employee {
```

```
protected String name; protected long salary;
```

```
public Employee () {
```

```
}
```

```
public Employee (String name) { this.name = name;  
}
```

```
public abstract void caculateSalary();
```

```
protected String getEmployeeType ();
```

```
public void printInfo() {
```

```
System.out.println ("===== Employee:" + name + "====="); System.out.println
```

```
("-Employee's type:" + getEmployeeType()); System.out.println ("- Salary:" + salary + "VND");
```

```
}
```

```
}
```

Employee subclasses will have a bit of constraints, I will go into detail step by step to build *FulltimeEmployee* class for you to see. When you have just declared this class inheriting from *Employee* , you will see the system error with a bulb-shaped icon next to the red cross (I have mentioned this type of Eclipse error in this *lesson* , you please refer).



To fix this is very easy, you just need to *override* the *Abstract* methods from the *Employee* class . If you don't know how many *Abstraction* methods need to *override* , just hover your mouse pointer over that bulb icon and you will see it listing all the methods you need.



Or click completely on the light bulb, you will see a hint. Now choose *Add unimplemented methods* .



After selecting the above option, the system will create all methods to *override* the *Abstraction* methods from *Employee* . And when this system also no longer reports errors. If *FulltimeEmployee* inherits from *Employee* , without implementing all the *Abstraction* methods from *Employee* , then you will never be able to execute the application.



And here is the final code of the *FulltimeEmployee* and *ParttimeEmployee* class

```
package model;
import util.Configs;

/** 
 * FulltimeEmployee is a full-time employee */

public class FulltimeEmployee extends Employee {
    private int oTDays; // The employee's overtime day
    private int position; // The position is normal employee or bos

    public FulltimeEmployee (String name, int oTDays) {
        super (name);
        this.oTDays = oTDays;
        this.position = Configs.EMPLOYEE_NORMAL; // The default is normal
        employee
    }

    public void setPosition (int position) { this.position = position;
    }

    @Override
    public String getEmployeeType () {
        if (position == Configs.EMPLOYEE_NORMAL) {

            return " Fulltime Employee " + (oTDays > 0? "with overtime": "");
        } else {
            return " Fulltime Boss " + (oTDays > 0? "(with overtime)": "");
        }
    }

    public void caculateSalary () {
```

```

if (position == Configs.EMPLOYEE_NORMAL) {
    salary = Configs.SALARY_NORMAL_FULLTIME + oTDays *
    Configs.MONEY_EACH_OT_DAY; } else if (position ==
Configs.EMPLOYEE_BOSS) { salary = Configs.SALARY_BOSS_FULLTIME
+ oTDays * Configs.MONEY_EACH_OT_DAY; }
}
}

package model;
import util.Configs;

/***
 * ParttimeEmployee
 */

public class ParttimeEmployee extends Employee {
private int workHours; // Total working hours of employee

public ParttimeEmployee (String name, int workHours) { this.name = name;
this.workHours = workHours;

}

@Override
public String getEmployeeType () { return "Parttime Employee"; }

public void caculateSalary () {
salary = Configs.SALARY_PARTIME_EACH_HOUR * workHours;
}
}

```

The code in the main() method will still be the same as in lesson 30.

```

package main;
import java.util.Scanner;
import model.Employee;
import model.FulltimeEmployee; import model.ParttimeEmployee; public class
MainClass {

public static void main (String [] args) {
// Ask user to enter number of employee
Scanner scanner = new Scanner (System.in);
System.out.print ("Please enter the number of employees:"); int

```

```

numberEmployees = Integer.parseInt (scanner.nextLine ());

// Declare an array of employee
Employee [] arrayEmployees = new Employee [numberEmployees];
for (int i = 0; i < numberEmployees; i ++) {

// Declare each type of employee, and ask the user to enter employee information
System.out.print ("Employee name" + (i + 1) + ":");
String name = scanner.nextLine ();
System.out.print ("Are employees (1-Full-time; 2-Part-time):");
int isEmployee = Integer.parseInt (scanner.nextLine ());
if (isEmployee == 1) {

// Full time employee
System.out.print ("Employee position (1-Boss; 2-Normal Employee):");
int position = Integer.parseInt (scanner.nextLine ());
System.out.print ("Overtime (if any):");
int oTDays = Integer.parseInt (scanner.nextLine ());
arrayEmployees [i] = new FulltimeEmployee (name, oTDays, position);

} else {
System.out.print ("Working hours:");
int workHours = Integer.parseInt (scanner.nextLine ());
arrayEmployees [i] = new ParttimeEmployee (name, workHours);

}
}

System.out.println ("\nSalary result \ n");

// Caculate employee 'salary and print employee information for (Employee
employee: arrayEmployees) {
employee.caculateSalary ();
employee.printInfo();
}
}
}
}

```

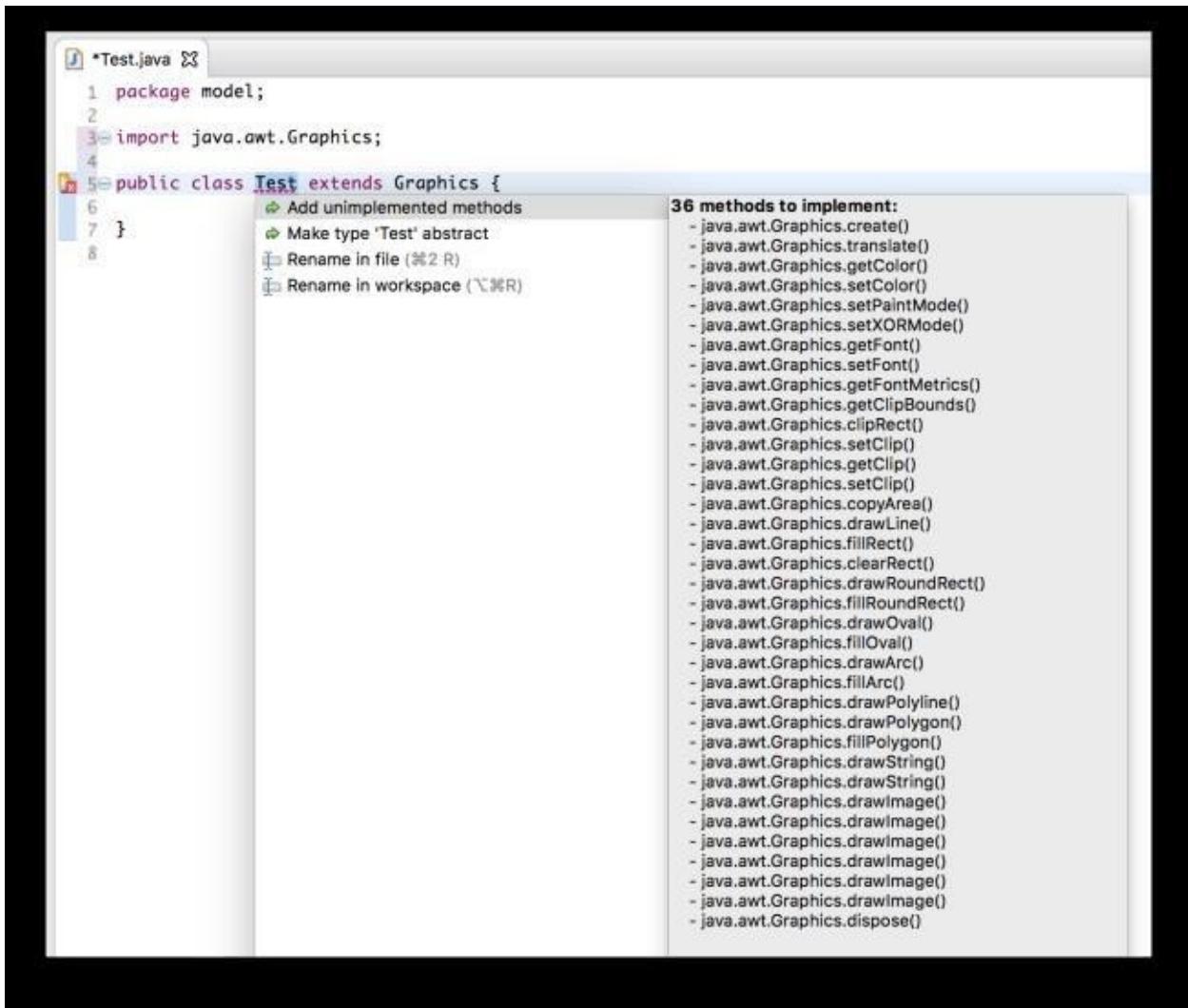
Through this exercise, you have understood the purpose and way of binding of an *Abstract* class , right? In this final section, I will show you some of the *Abstraction* classes built into the Java library, so that you have a broader

understanding of this knowledge.

2. Experience Some Abstract Layers Of Java Platform

This experiment is *only to verify some of the Abstraction classes available from the system* . We do not always have to inherit from these classes, unless you want to *override* some of its methods to extend beyond the capabilities of the original class, which is called customizing an object. You will encounter more of this system's *Abstraction* classes customizing when *building Android apps* .

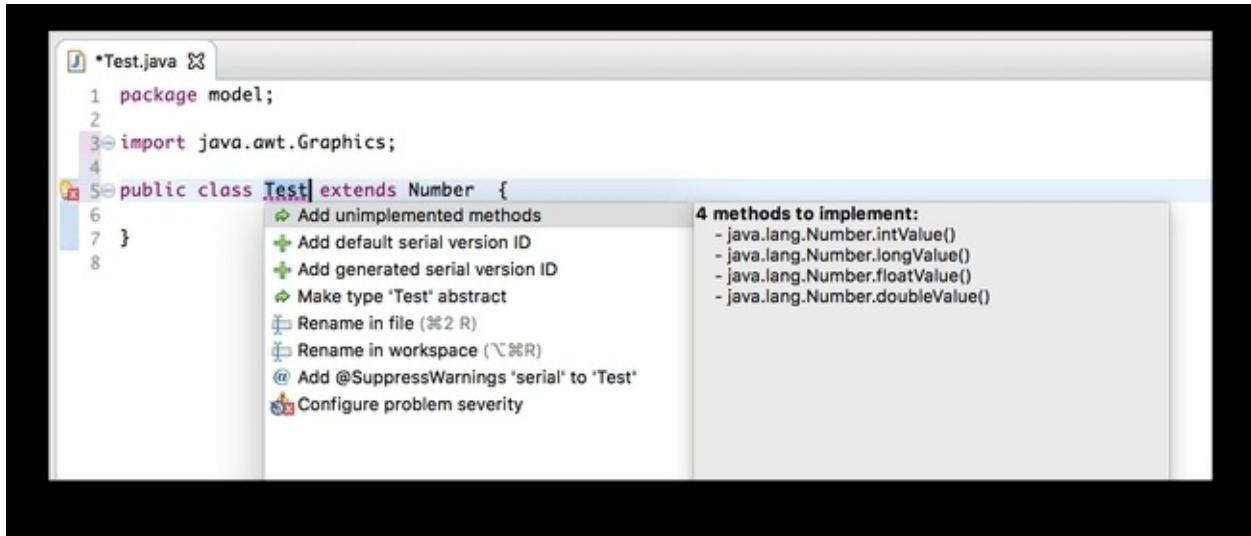
First Experience. If you system's *Graphics* class , of *Abstraction* methods from the class. try to build a class and then inherit from the you'll get the "call" to implement dozens



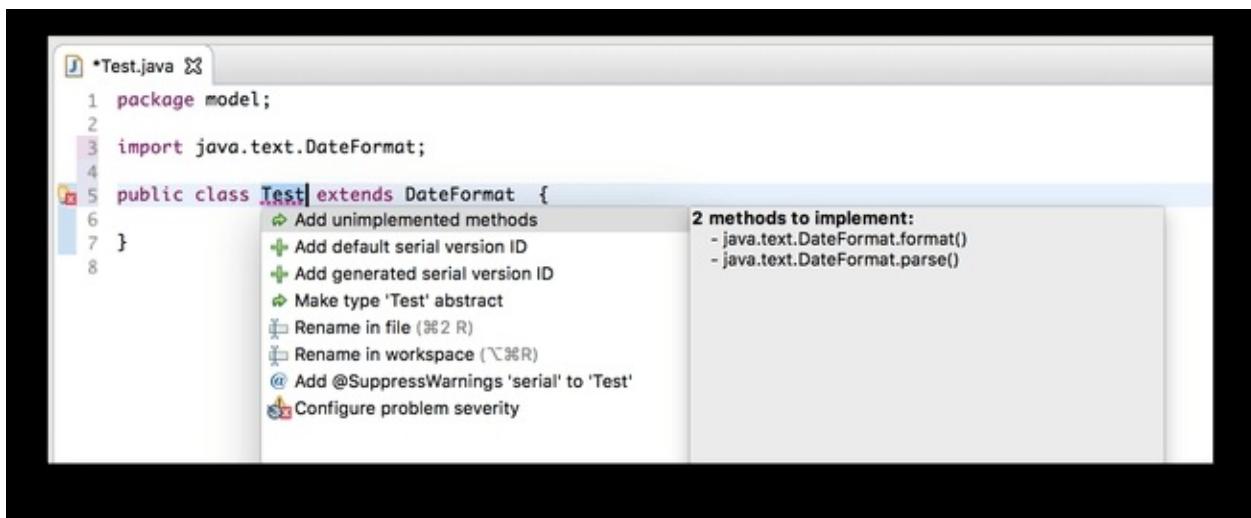
The screenshot shows an IDE interface with a code editor window titled "Test.java". The code contains a single class definition:1 package model;
2
3 import java.awt.Graphics;
4
5 public class Test extends Graphics {
6 //...
7 }
8A code completion tooltip is displayed over the "Test" class definition, listing 36 methods to implement. The methods listed are:

- java.awt.Graphics.create()
- java.awt.Graphics.translate()
- java.awt.Graphics.getColor()
- java.awt.Graphics.setColor()
- java.awt.Graphics.setPaintMode()
- java.awt.Graphics.setXORMode()
- java.awt.Graphics.getFont()
- java.awt.Graphics.setFont()
- java.awt.Graphics.getFontMetrics()
- java.awt.Graphics.getClipBounds()
- java.awt.Graphics.clipRect()
- java.awt.Graphics.setClip()
- java.awt.Graphics.getClip()
- java.awt.Graphics.setClip()
- java.awt.Graphics.copyArea()
- java.awt.Graphics.drawLine()
- java.awt.Graphics.fillRect()
- java.awt.Graphics.clearRect()
- java.awt.Graphics.drawRoundRect()
- java.awt.Graphics.fillRoundRect()
- java.awt.Graphics.drawOval()
- java.awt.Graphics.fillOval()
- java.awt.Graphics.drawArc()
- java.awt.Graphics.fillArc()
- java.awt.Graphics.drawPolyline()
- java.awt.Graphics.drawPolygon()
- java.awt.Graphics.fillPolygon()
- java.awt.Graphics.drawString()
- java.awt.Graphics.drawString()
- java.awt.Graphics.drawImage()
- java.awt.Graphics.drawImage()
- java.awt.Graphics.drawImage()
- java.awt.Graphics.drawImage()
- java.awt.Graphics.dispose()

This is the same if you inherit from *Number* class .



Or with the *DateFormat* class and with many other classes on the Java platform.



That's all, enough for you to see that the Java platform has built in a lot of *Abstraction* classes . Later if there is any need to *implement* methods when you inherit a strange class, then you know that class is the *Abstract* class . By examining some of the *Abstraction* classes from such systems, we have concluded this's lesson as well.

Lesson 31: Nested Classes

Reading to the title, you probably already know the content of the lesson. It is

the knowledge of declaring a class within another. This is not a parent-child relationship, so it is not inheritance, it is just a class declared inside another class only. So let's find out what nested classes is and why do it so.

I. What Is Cage Class?

I just repeat the above point, *nested class is the declaration of a class inside another class* .

Class that contains other classes inside it is called *Outer Class* , which can be understood in Vietnamese as *Bao* class .

And the class inside the *Outer Class* is divided into two different categories.

- One is *a non-static class (non-static class)* , it is called *Inner Class* .
- The other type is *a static class (static class)* , it is called *Static Nested Class* .

Wait to talk about the role of each type of *Inner Class* and *Static Nested Class* , let's come to the declaration syntax of each type as follows.

This is the syntax of an *Inner Class* .

```
class OuterClass {  
...  
class InnerClass {  
...  
}  
}
```

Here is the syntax of a *Static Nested Class* .

```
class OuterClass {  
...  
static class StaticNestedClass { ...  
} }
```

The two syntaxes are nothing but *static* keyword . Through the above syntax, we have the following ideas to keep in mind.

However, the syntax has only one *Outer Class* containing an *Inner Class* or an internal *Static Nested Class* . But you should know that within an *Outer Class* can contain multiple *Inner Class* , many *Static Nested Class* , or contain both *Inner Class* and *Static Nested Class* .

- And although the syntax only refers to *nested classes* , but you can nested *interfaces* in classes, or nested *interfaces* together, or nested classes into

interfaces .

- *Inner class* is now considered a member of *The Outer Class* , so you can specify it for accessibility , like *private* , *public* , *protected* , or *default* (ie no accessibility declaration). This is different from the *Outer Class* or classes that you are familiar with, which can only be declared *public* or *default* (this declaration only allows classes in the same *package* to be seen) only.

- Practice Declaring & Using Inner Class

Keep in mind that the application of nested classes is quite a lot, but whether or not to use them is optional, and if needed, it is not difficult.

In this exercise, we will see how to define and use an *Inner Class* . After this exercise we will talk more specifically about the uses of the nested class later.

The following code declares class *Coordinate* which is placed inside *Geometry* class . You can also declare a *Coordinate* class outside of *Geometry* as you did in some lesson, but by putting *Coordinate* in *Geometry* , you can see that this class can use the *name* property of the *wrapper* class.

```
public class Geometry {  
    public static final float PI = 3.14f;  
    public String name;  
    public Coordinate coordinate;  
  
    // Constructor  
    public Geometry(int x, int y) {  
        this.name = "Geometry";  
  
        this.coordinate = new Coordinate(); this.coordinate.x = x;  
        this.coordinate.y = y;  
    }  
    public class Coordinate {  
        int x; int y;  
  
        public void printInfo() {  
            System.out.println("Shape: " + name);  
            System.out.println("Coordinate: x = " + x + "; y = " + y);  
        }  
    }  
}
```

}

The nested class using the properties of the wrapper class will be more clearly stated in the section below the exercises.

Now, continue to see in the *main () method* we declare the *Geometry* class and call *printInfo ()* of *Coordinate* as follows, what will happen?

```
public class MainClass {
```

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

```
        Geometry geometry = new Geometry(10, 20); Geometry.coordinate.printInfo();
```

```
}
```

```
}
```

We will receive the following information.

Shape: Geometry

Coordinate: x = 10; y = 20 With the above code of the *main () method* , you can see that we declare the object of the *Geometry* class , and then use its variable *xeDo* , this *Geometry.Coordinate.printInfo ()* . You can also declare the *Coordinate* class to use for the next time, as follows.

```
public class MainClass {
```

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

```
        Geometry geometry = new Geometry(10, 20); Geometry.Coordinate coordinate  
        = geometry.new Coordinate(); Coordinate.printInfo();
```

```
}
```

above written like a private object of

```
}
```

Do you see how to declare the *Coordinate* class like the above code is strange but familiar? We have approach this in Practice#2 Lesson 18

“ Circle circle = new MyFirstClass(). new Circle();”

You may understand of *Geometry* . Therefore objects *Geometry* is *Geometry* only. at this point that *Coordinate* is considered a member you can only initiate independent *Coordinate* through

When you execute the above line of *main ()* , you will get a slightly different output as follows. Can you tell yourself why the results were so different?

Shape: Geometry

Coordinate: x = 0; y = 0

- Practice Declaring & Using Static Nested Class

I will also use the script from the above exercise, but replace the *Coordinate* class inside *Geometry* with a *Static Nested Class*. See if there is any difference.

```
public class Geometry {  
  
    public static final float PI = 3.14f; public static String name; public Coordinate  
    Coordinate;  
  
    // Constructor  
    public Geometry(int x, int y) {  
        this.name = "Geometry";  
  
        this.coordinate = new Coordinate(); this.coordinate.x = x;  
        this.coordinate.y = y;  
  
    }  
    public static class Coordinate {  
        int x; int y;  
  
        public void printInfo() {  
            System.out.println("Shape: " + name);  
            System.out.println("Coordinate: x = " + x + "; y = " + y);  
  
        }  
    }  
}
```

As you can see, since *Coordinate* is defined as a *static* class , it can only access *static* methods and properties of the *bound* class. This is similar to *the static method* that you have learned, then the *static* method can only call the properties that are declared *static* .

With the declaration *Coordinate* like this, you still call and use objects through which *Geometry* normally.

```
public class MainClass {
```

```
public static void main(String[] args) {  
    Geometry geometry = new Geometry(10, 20); geometry.coordinate.printInfo();  
}  
}
```

} Or you can directly *declare Coordinate* slightly differently than when declaring *Coordinate* as *Inner Class* as above.

```
public class MainClass {  
  
    public static void main(String[] args) {  
        Geometry.Coordinate coordinate = new Geometry.Coordinate();  
        coordinate.printInfo();  
    }  
}
```

} Try executing the application yourself to see what the console prints.

II. When Should I Use The Cage & Its Use?

Based on what we are familiar with with the nested class above, we have had enough experience to sum up some of its uses.

- If you have a class *A* only to a class *B* area. Ie *B* no one except the user to a stop. Then you might consider organizing way *B would be incorporated into the class A class* . For the benefits of this, you can see the following ideas.
- The nesting of classes increases data *encapsulation* . For example, if class *B* is in class *A* , you can declare *B* as *private* , then besides *A* , there is no other class that can know the existence of *B* and its values. can say you did "*Concealed*" *B* from the world, except *A* .
- In the opposite direction. Nested class *B* is accessible to all members (properties and methods) of wrapper class *A* , even if the members of *A* are declared *private* . Of course, when nested classes like that, your code will be easier to read (because you don't have to find and open too many classes), resulting in easier maintenance.

- **Exercise Number 1**

We test to see if we understood the lesson through this exercise.

Assume the following wrapper and nested classes are defined. public class

```
OuterClass {  
  
    public void show() { InnerClass innerClass = new InnerClass();  
    innerClass.show();  
  
    }  
    public class InnerClass {  
        public void show() {  
            System.out.println("This is inner class"); }  
        }  
    }
```

You try to answer to see what the output will be to the console with the code in *main () method* as follows.

```
public class MainClass {  
  
    public static void main(String[] args) { OuterClass outerClass = new  
    OuterClass(); outerClass.show();  
  
    InnerClass innerClass = outerClass.new InnerClass(); innerClass.show();  
    }  
}
```

This is inner class

This is inner class

- Exercise Number 2

Similar to *exercise 1* . Suppose you have the following definitions of wrappers and nested classes.

```
public class DateMonthYear {  
    public int date, month, year;  
    public class HourMinuteSecond { public int hour, minute, second;  
  
    public void printInfo() {  
        System.out.println("Date: " + date + "/" + month + "/" + year);  
        System.out.println("Time: " + hour + ":" + minute + ":" + second);  
  
    }  
}
```

And then in the *main () method* I call them as follows.

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

```
DateMonthYear date = new DateMonthYear(); date.date = 31;  
date.month = 10;  
date.year = 2017;  
  
HourMinuteSecond time = date.new HourMinuteSecond(); time.hour = 10;  
time.minute = 15;  
time.second = 30;  
  
time.printInfo(); }  
}
```

Try to answer what the output is to the console. And here is the answer.

Date: 31/10/2017

Time: 10:15:30

That is only the knowledge of nested classes. You will get used to using this nested class when using a lot of the Java language, or when you read other people's code, or when *programming with Android* .

Lesson 32: Anonymous Class

This we are going to come up with a more interesting way to use classes and objects inside Java. Through the lesson, you will learn that it is not always necessary to declare an explicit class and then use it through object declarations from that class. You can create comfortable objects out of a class without anyone knowing which class the object belongs to.

After the lesson, you will know how to use classes in Java in a more flexible, concise manner, and if you look familiar, it will be easier to understand. We invite you to join us in this's lesson.

I. What Is Anonymous Class?

The Anonymous Class , in English called the *Anonymous Class* , or more often referred to with its full name, the *Anonymous Inner Class* .

Of course a class is called *Anonymous* means it will not have a specific name. The *Anonymous* class will attach inheritance (including inheritance from a *normal parent* class or *abstract* superclass), and attach to the *interface as well*. And because the *Anonymous* class is also called *Anonymous Inner Class* , it also

has a little bit of knowledge about *nested classes* .

So let's find out what the *Anonymous* class really is and involve so much knowledge.

II. Identity Class *Anonymous*

You can see that this section is talking about the *syntax* of an *Anonymous* class . It will help you identify what is a normally declared class, and what is an *Anonymous* class . And since *Anonymous* classes don't have a certain thing in common, we're not talking about syntax, but about recognition.

You all know, to create a certain class, you have to declare that class with the *class* keyword , obviously, then you *have to name it* , like what you learned in *the first lessons* about OOP .

`class_class { properties; methods;` From the class declaration above, you will create objects of the class with the keyword *new* later.

As for *Anonymous* classes , you will *not need to declare a class, but can still create its objects* . It sounds vague, without the class, where the object will be created.

First of all, please see the following "syntax" of an *Anonymous* class . By writing the following, you will create an object named *employee* , so this object will be an object created from any class, is the class *Employee* ?

```
Employee employee = new Employee () { properties; methods;  
}
```

I would like to say an *Anonymous* class, somewhere. Class *Nameless* above example is what certain class is command block with *properties* and *methods* , class *Anonymous* This before that, the *Employee* class like

the *Employee* class will be the above code is not declared normally

defined by the is a subclass of *employee* , then layer *Nameless* are assigned to the object *employee* . You can freely use this *employee* object *later* on.

In conclusion, if you ask what class the *employee* is from, the answer can only be *Anonymous*.

If you still do not fully understand the concept of *Anonymous*, you can read more below, I will present more specifically.

III. When Should Anonymous Classes Be Used?

As the above points, I would like to summarize. The *Anonymous* class is often used when you do not want to specifically declare the subclass of a certain class (including the abstract class and the ordinary class you know), even when you do not want to specifically declare the implementation class. declarations of a certain interface, but still want to use their objects.

Talking around in circles is just that much. Here I would like to give an example of using the *Anonymous* class to make it easier for you to understand.

- Anonymous Class Building Practice

This exercise helps you to understand what an *Anonymous* class is, through a built-in interface called *Geometry*.

I would like to take back the *practice of declaring an interface* that you know to create an interface with the name *Geometry* first, this code has nothing to do with the *Anonymous* class.

```
interface Geometry {  
    float PI = 3.14f;  
    void enterRadius(float radius);  
    float calculatePerimeter();  
    float calculateArea();  
    void printInfo(); }
```

With the interface *Geometry* above, if used in a normal way, you can declare a class "*Huu Danh*" named *Circle*. This *Circle* class implements the abstract methods from the *Geometry* interface as follows.

```
public class Circle implements Geometry {  
    protected String name; protected float radius;  
  
    // Constructor  
    public Circle(float radius) { this.name = "Circle"; }
```

```

this.radius = radius; }

@Override
public void enterRadius(float radius) { this.radius = radius;
@Override
public float calculatePerimeter() { return 2 * PI * radius;
}
@Override
public float calculateArea() { return PI * radius * radius; }

@Override
public void printInfo() {
System.out.println(name);
System.out.println("Perimeter: " + calculatePerimeter());
System.out.println("Area: " + calculateArea()); }

}

```

Continuing, if we want to use the above *Circle* class , we will declare the object and call it as usual.

```

public static void main(String[] args) { Circle circle = new Circle(10);
circle.printInfo();

}

```

The fact that you declare a class named *Circle* , then you have declared a class with a name absolutely.

Now we come to how to use declare any *Circle* class , but you superclass. So, forget about the statement that declared the class *Circle* above, but use it now and always the object is initialized in the following code.
the *Anonymous* class . At first, you don't need to still need the *Geometry* interface to be the base

```

public static void main(String[] args) {
Geometry randomShape = new Geometry() { protected float radius;
@Override
public void enterRadius(float radius) { this.radius = radius;
}
@Override
public float calculatePerimeter() { return 2 * PI * radius;
}

```

```
}
```

@Override
public float calculateArea() { return PI * radius * radius; }

```
@Override  
public void printInfo() {  
    System.out.println("Nameless Shape");  
    System.out.println("Perimeter: " + calculatePerimeter());  
    System.out.println("Area: " + calculateArea()); }  
};
```

```
randomShape.enterRadius(10); randomShape.printInfo(); }
```

With the above code, you must pay close attention, the side effects of the code will dazzle those who are not familiar with.

You see, with the declaration *Geometry randomShape = new Geometry {...}* , this declaration will obviously create an object *randomShape* , but you know that *randomShape* is not created from *Geometry* right. It's very easy to explain, because *Geometry* is an interface, you already know an interface cannot be used to create an object!!!

So *randomShape* is an object created from an *Anonymous* class , this *Anonymous* class has fully implemented the interface *Geometry* through its block. The following lines of code in the above method are the uses of the *randomShape* object to enter the information and output it to the console to produce the same results as when not using the *Anonymous* class above. You try it out.

IV. Class Anonymous

You have gotten to know and understand the *Anonymous* class well , right? In order to easily detect and apply the *Anonymous* class to reality, I would like to summarize the classification of the *Anonymous* class .

1. *Anonymous Classes Created Through Inheritance From Another Class*

If you have a class, whether it's an ordinary class or an abstract class. Then instead of declaring a subclass of it with a specific name, you can declare a subclass of *Anonymous* . You need to inherit the *Thread* class from the system as follows.

```

public static void main(String[] args) {
// Example of Anonymous class created // through inheriting from Thread class
Thread t = new Thread() {

@Override
public void run() {

System.out.println("You can test this code");
};

t.start();

}

```

This example is probably not difficult for you. The object *t* that you declare is not an object of *Thread* class , but just an object of *Anonymous* class inheriting from *Thread* .

2. Anonymous Classes Created Through Implementation From Another Interface

This section is similar to the example in the above lesson. Since you have an interface, instead of declaring an explicit implementation from this interface, you can apply the *Anonymous* class .

The following example will deploy from an interface named *Runnable* . You will also get acquainted with this code in the lesson about *multi-threaded* later.

```

public static void main(String[] args) {

// The Anonymous class is created
// through implementation from Interface Runnable
Runnable r = new Runnable() {

@Override
public void run() {

System.out.println("You can test this code ");
};

Thread t = new Thread(r);
t.start();

}

```

3. Anonymous Class Is Used As A Pass Parameter

The example below shows that instead of passing *r* to *Thread ()'s constructor* ,

```

you can create the Anonymous class as a parameter instead of the r declaration .
public static void main(String[] args) { // The Anonymous class is created // as a
parameter passed
Thread t = new Thread(new Runnable() {
@Override
public void run() {

System.out.println("You can test this code"); }
});
t.start();

}

```

V. Anonymous Class Characteristics

Below we will summarize some memorable features of the *Anonymous* class . And since the *Anonymous* class will be used quite a bit due to its usability, you should try to keep these characteristics in mind as well.

- If a normal class can implement as many interfaces as you like, then *Anonymous class can only deploy from only one interface* .
- If a normal class can both inherit from a certain class and be able to deploy from many other interfaces, then *Anonymous class can only either inherit or implement another class or interface* .

With a normal class, you can define arbitrary *constructors* . But the *Anonymous class does not have any constructors* . This is easy to understand, because the constructor must have the same name as the class, but the *Anonymous* class has no name, so you can never define a constructor for it.

And since an *Anonymous* class is declared within another class, it has a similarity to *Nested class* , in that it can access the members of its wrapper class.

Above is the knowledge related to the *Anonymous* class . Hoping to finish reading this lesson, many of you will shout that so far the uses of classes of this type have been called with such a name. Because I used to be like that.

Lesson 33: Wrapper Class

This's lesson takes us back to the past, to add more knowledge that was quite "rustic" in a lesson at that time. The past is here when we are still familiar with Java *primitive data types*. By now you know Java has *8 primitive types*, they include *int*, *short*, *long*, *byte*, *float*, *double*, *char* and *boolean*. You can review full *lesson # 4* for more information about these data types.

This you will be familiar with classes called *Wrapper*. Let's see what these classes help to add to the primitive data types listed above.

I. Familiar With *Wrapper* Class

Wrapper class is actually just a generic name for many different classes. Since all the classes in this's lesson have the same function, we call it the same *Wrapper* name .

And because as I said, Java has *8 primitive data types*, so there will also be *8 Wrapper classes for each of these primitives*. These include.

- The *Byte* class is a *Wrapper* class for the *byte* data type .
- The *Short* class is a *Wrapper* class for the *short* data type .
- Class *Integer* is a class *wrapper* for the data type *int* .
- The *Long* class is a *Wrapper* class for the *long* data type .
- Class *Float* class *wrapper* for the data type *float* .
- The *Double* class is a *Wrapper* class for the *double* data type .
- The *Character* class is a *Wrapper* class for the *char* data type .
- Class *Boolean* is a *Wrapper* class for *boolean* data type .

As you can see, each *Wrapper* class for each primitive data type will be an object that wraps the primitive type it supports. In other words, each *Wrapper* class will contain a primitive type within it. And, the *Wrapper* class helps in constructing other methods that complement the original simplicity of the primitive type.

One thing you can keep in mind is that, just as with your knowledge of *Strings* you already know, *Wrapper* classes English . You can see are also classes

a little bit more of *immutable* value , called *Immutable* in knowledge about the *Immutable* in Lesson 13. Additionally, the class *Wrapper* is the *class final* , and so you can not

create subclasses of them.

II. Why Use Wrapper Class?

First of all, the most fundamental thing about this is, *Wrapper* classes help us *convert between a primitive data type to an object data type, and vice versa* . You can see examples for using primitive data types and its *Wrapper* type as follows.

```
int a = 20; // a is a variable of primitive data type int  
Integer i = Integer.valueOf(a); // i is a variable of data type Integer, // created  
from the primitive variable a  
As you can see, variable a is of type int , and variable i is of type Integer .
```

The next point to why this question is, if with primitive data types, you have only one option: to create a variable and then use its value (if you don't assign a value, it will still create a variable. default values, you can review *the primitive data types*). As for object types, its default value is *null* , this *null* value can be utilized in some cases, such as having no meaning. In addition, object types also carry many useful methods, enriching the applicability of the data type.

In addition, some other constructs within the Java language, such as the constructs for lists that we will become familiar with, such as *ArrayList* or *Vector* contain collections of object data types instead of primitives, so knowing and manipulating *Wrapper* classes is a must.

In addition, the object data type will be more suitable for implementing *multithreading* and *synchronization*, which we will discuss in another lesson.

III. Switching Between Primitive and Wrapper

1. Convert Original Style to Wrapper Style

Converting a primitive to its *Wrapper* style is known as *Boxing* . Not meant for boxing. Boxing here is meant to be *boxed* , which means packing the original data into its *Wrapper* box . Like the example you saw above, when an *int a* is converted to an *Integer i* .

You can do boxing through the *Wrapper constructors* .

```
// Các dạng Boxing int a = 500;  
Integer i = new Integer(a);  
Integer j = new Integer(500); Float f = new Float(4.5);  
Double d = new Double(5);  
Character ch = new Character('a'); Boolean b = new Boolean(true);
```

Or it is possible to directly assign primitive values to *Wrapper* classes , this way is also known as *Autoboxing* , which means the system will convert automatically.

```
// Các dạng Autoboxing int a = 500;  
Integer i = a;  
Integer j = 500;  
Float f = 4.5f;  
Double d = 5d;  
Character ch = 'a';  
Boolean b = true;
```

```
// Đây cũng là một dạng Autoboxing mà bạn sẽ được biết khi học đến bài về  
Collection ArrayList<Integer> arrInt = new ArrayList<Integer>();  
arrInt.add(25);
```

2. Convert *Wrapper* to Original Style

Contrary to the above, when you switch from a *Wrapper* type to its primitive it is called *Unboxing* , which means opening the box, ie opening the *Wrapper* box to get the primitive data out.

You can perform unboxing through the *xxxValue ()* methods . With *xxx* represents each type of data, such as the following example.

```
int a = 500;  
Integer i = a; // Autoboxing int i2 = i.intValue(); // Unboxing  
  
Integer j = 500; // Autoboxing int j2 = j.intValue(); // Unboxing  
  
Float f = 4.5f; // Autoboxing  
float f2 = f.floatValue(); // Unboxing Double d = 5d; // Autoboxing  
double d2 = d.doubleValue(); // Unboxing
```

```
Character ch = 'a'; // Autoboxing char ch2 = ch.charValue(); // Unboxing  
Boolean b = true; // Autoboxing  
boolean b2 = b.booleanValue(); // Unboxing
```

```
ArrayList<Integer> arrInt = new ArrayList<Integer>(); arrInt.add(25); //  
Autoboxing  
int arr0 = arrInt.get(0).intValue(); // Unboxing
```

Similar to autoboxing, the unboxing technique can also be written like this.

```
int a = 500;  
Integer i = a;  
int i2 = i; // Unboxing
```

```
Integer j = 500;  
int j2 = j; // Unboxing  
Float f = 4.5f;  
float f2 = f; // Unboxing  
Double d = 5d;  
double d2 = d; // Unboxing  
Character ch = 'a';  
char ch2 = ch; // Unboxing  
Boolean b = true;  
boolean b2 = b; // Unboxing
```

```
ArrayList<Integer> arrInt = new ArrayList<Integer>(); arrInt.add(25);  
int arr0 = arrInt.get(0); // Unboxing
```

IV. Helpful Methods Of Wrapper Class

As above, I said that *Wrapper* classes help create primitive data types that only contain data, becoming an object with more useful methods. Here I will list their useful methods for your reference.

1. parseXxx()

The parameter passed to this *static* method is a string, the result is a primitive value corresponding to the string passed.

```
int i = Integer.parseInt("10");  
float f = Float.parseFloat("4.5");
```

```
boolean b = Boolean.parseBoolean("true");  
System.out.println(i); System.out.println(f); System.out.println(b);
```

You can also guess the output of the console printout of the code above, right?

2. *toString ()*

Unlike the *toString ()* a *static* method , it has to that *Wrapper* class , and the result is a string corresponding to the value passed. This example experiments with *Integer* class , other *Wrapper* classes will be similar.

of the Object class , this *toString ()* of *Wrapper* classes is a value passed as the primitive data type corresponding

```
String sI = Integer.toString(10); System.out.println(sI);
```

The output to the console is the string "10" .

3. *xxxValue ()*

You are familiar with this method over and over. Specifically, methods of this type help to convert a value of a *Wrapper* class to the primitive data type (unboxing). Sometimes this method also helps to convert the data type as if you *explicitly cast* a value. You can understand more easily by looking at the following example.

```
Double d = 50.5; int i = d.intValue(); byte b = d.byteValue();
```

```
System.out.println(d); System.out.println(i); System.out.println(b);
```

The output to the console is 50.5 , 50, and 50 respectively .

4. *compareTo ()*

If you remember, in the string lesson we also covered the *compareTo () method* used to compare string values against each other. So for the *Wrapper* classes that we are familiar with this, its functionality is still fully applied. That is, this method will be used to compare two values of two *Wrapper* classes (of the same data type) with each other.

Specifically, with you calling *lopWrapper1.compareTo (lopWrapper2)* , the result will be as follows.

- If *the method's result returns a negative number* , then *lopWrapper1* will have a smaller value than *lopWrapper2* .

- If the method's result returns 0 , then *lopWrapper1* will be equal to *lopWrapper2* .
- If the method's result returns a positive number , *lopWrapper1* will have a greater value than *lopWrapper2* .

You can see the following example for better understanding.

```
Integer i = 50;  
Integer i1 = Integer.parseInt("50"); Integer i2 = Integer.valueOf(52); Integer i3 =  
30;
```

```
System.out.println("CompareTo i & i1: " + i.compareTo(i1));  
System.out.println("CompareTo i & i2: " + i.compareTo(i2));  
System.out.println("CompareTo i & i3: " + i.compareTo(i3));
```

The output to the console will be.

```
CompareTo i & i1: 0  
CompareTo i & i2: -1 CompareTo i & i3: 1
```

5. *compare ()*

This method has the same usage and usage as *compareTo ()* above. The difference is that this is the *static* method of each *Wrapper* class , so you can call it directly from the class. At the same time, the parameter passed are two values of the two *Wrapper* classes . The return result of *compare ()* also has one of three values (*negative* , 0 , *positive*) as with *compareTo ()* above.

```
Integer i1 = Integer.parseInt("50"); Integer i2 = Integer.valueOf(52);  
System.out.println("Compare i1 & i2: " + Integer.compare(i1, i2));  
Float f1 = new Float("20.25f"); Float f2 = new Float("2.43f");  
System.out.println("Compare f1 & f2: " + Float.compare(f1,f2));
```

The output to the console is.

```
Compare i1 & i2: -1  
Compare f1 & f2: 1
```

6. *equals ()*

Similar to *equals ()* when comparing strings. This method will compare the values of *Wrapper* classes and return a *boolean* type , where *true* is equal and *false* is different. Like the following example.

```
Integer i1 = Integer.parseInt("50"); Integer i2 = Integer.valueOf(50);
System.out.println("Compare i1 & i2: " + i1.equals(i2));
Float f1 = new Float("20.25f"); Float f2 = new Float("2.43f");
System.out.println("Compare f1 & f2: " + f1.equals(f2)); The output to the
console is.
```

Compare i1 & i2: true

Compare f1 & f2: false

- Exercise Number 1

Try executing the following code, and try to find out why the data printed to the console is like that.

```
int i = Integer.parseInt("10");
float f = Float.parseFloat("4.5a");
System.out.println(i);
System.out.println(f);
```

- Exercise Number 2

Please indicate the output of the console printout of the following line of code.

```
int i = Integer.parseInt("10.5");
System.out.println(i);
```

- Exercise Number 3

In addition to the useful methods of the *Wrapper* classes that I have listed above, please find out about other methods for yourself, they are as useful as what I mentioned. Such as.

- Static methods of *Integer , Long , Float , and Double classes* : *toHexString()* , *toOctalString ()* , *toBinaryString ()* , *max ()* , *min ()* , ...
- Static methods of *Character class* : *isLowerCase()* , *isUpperCase()* , *isDigit()* , *toLowerCase ()* , *toUpperCase ()* , ...

So through this's lesson, you have learned how primitive data types in Java were "*surrounded*" by *Wrapper* classes , to make them more diverse and convenient. is not.

This's lesson takes us back to the past, to add more knowledge that was quite "*rustic*" in a lesson at that time. The past is here when we are still familiar with Java *primitive data types* .

By now you know Java has *8 primitive types* , they include *int , short , long , byte , float , double , char* and *boolean* . You can review full *lesson # 4* for more information about these data types.

This you will be familiar with classes called *Wrapper* . Let's see what these classes help to add to the primitive data types listed above.

V. Familiar With Wrapper Class

Wrapper class is actually just a generic name for many different classes. Since all the classes in this's lesson have the same function, we call it the same *Wrapper* name .

And because as I said, Java has *8 primitive data types* , so there will also be *8 Wrapper classes for each of these primitives* . These include.

- The *Byte* class is a *Wrapper* class for the *byte* data type .
- The *Short* class is a *Wrapper* class for the *short* data type .
- Class *Integer* is a class *wrapper* for the data type *int* .
- The *Long* class is a *Wrapper* class for the *long* data type .
- Class *Float* class *wrapper* for the data type *float*
- The *Double* class is a *Wrapper* class for the *double* data type .
- The *Character* class is a *Wrapper* class for the *char* data type .
- Class *Boolean* is a *Wrapper* class for *boolean* data type .

As you can see, each *Wrapper* class for each primitive data type will be an object that wraps the primitive type it supports. In other words, each *Wrapper* class will contain a primitive type within it. And, the *Wrapper* class helps in constructing other methods that complement the original simplicity of the primitive type.

One thing you can keep in mind is that, just as with your knowledge of *Strings* you already know, *Wrapper* classes English . You can see are also classes of *immutable* value , a little bit more knowledge about called *Immutable* in

the *Immutable* in *this lesson* . Additionally, the class *Wrapper* is the *class final* , and so you can not create subclasses of them.

VI. Why Use Wrapper Class?

First of all, the most fundamental thing about this is, *Wrapper* classes help us convert between a primitive data type to an object data type, and vice versa .

You can see examples for using primitive data types and its *Wrapper* type as follows.

```
int a = 20; // a is a variable of primitive data type int  
Integer i = Integer.valueOf(a); // i is a variable of data type Integer, // created  
from the primitive variable a  
As you can see, variable a is of type int , and variable i is of type Integer .
```

The next point to why this question is, if with primitive data types, you have only one option: to create a variable and then use its value (if you don't assign a value, it will still create a variable. default values, you can review *the primitive data types*). As for object types, its default value is *null* , this *null* value can be utilized in some cases, such as having no meaning. In addition, object types also carry many useful methods, enriching the applicability of the data type.

In addition, some other constructs within the Java language, such as the constructs for lists that we will become familiar with, such as *ArrayList* or *Vector* contain collections of object data types instead of primitives, so knowing and manipulating *Wrapper* classes is a must.

In addition, the object data type will be more suitable for implementing *multithreading* and *synchronization*, which we will discuss in another lesson.

VII. Switching Between Primitive and Wrapper

1. Convert Original Style to Wrapper Style

Converting a primitive to its *Wrapper* style is known as *Boxing* . Not meant for boxing. Boxing here is meant to be *boxed* , which means packing the original data into its *Wrapper* box . Like the example you saw above, when an *int a* is converted to an *Integer i* .

You can do boxing through the *Wrapper constructors* .

```
// Types of Boxing int a = 500;  
Integer i = new Integer(a);  
Integer j = new Integer(500); Float f = new Float(4.5);  
Double d = new Double(5);  
Character ch = new Character('a'); Boolean b = new Boolean(true);
```

Or it is possible to directly assign primitive values to *Wrapper* classes , this way is also known as *Autoboxing* , which means the system will convert automatically.

```
// Types of Autoboxing int a = 500;  
Integer i = a;  
Integer j = 500;  
Float f = 4.5f;  
Double d = 5d;  
Character ch = 'a';  
Boolean b = true;  
  
// A type of Autoboxing when you get to Collection ArrayList<Integer> arrInt =  
new ArrayList<Integer>(); arrInt.add(25);
```

2. Convert *Wrapper* to *Original Style*

Contrary to the above, when you switch from a *Wrapper* type to its primitive it is called *Unboxing* , which means opening the box, ie opening the *Wrapper* box to get the primitive data out.

You can perform unboxing through the *xxxValue ()* methods . With *xxx* represents each type of data, such as the following example.

```
int a = 500;  
Integer i = a; // Autoboxing int i2 = i.intValue(); // Unboxing  
  
Integer j = 500; // Autoboxing int j2 = j.intValue(); // Unboxing  
  
Float f = 4.5f; // Autoboxing  
float f2 = f.floatValue(); // Unboxing Double d = 5d; // Autoboxing  
double d2 = d.doubleValue(); // Unboxing  
  
Character ch = 'a'; // Autoboxing char ch2 = ch.charValue(); // Unboxing  
Boolean b = true; // Autoboxing  
boolean b2 = b.booleanValue(); // Unboxing  
  
ArrayList<Integer> arrInt = new ArrayList<Integer>(); arrInt.add(25); //  
Autoboxing  
int arr0 = arrInt.get(0).intValue(); // Unboxing
```

Similar to autoboxing, the unboxing technique can also be written like this.

```
int a = 500;
Integer i = a;
int i2 = i; // Unboxing

Integer j = 500;
int j2 = j; // Unboxing
Float f = 4.5f;
float f2 = f; // Unboxing
Double d = 5d;
double d2 = d; // Unboxing
Character ch = 'a';
char ch2 = ch; // Unboxing
Boolean b = true;
boolean b2 = b; // Unboxing
```

```
ArrayList<Integer> arrInt = new ArrayList<Integer>(); arrInt.add(25);
int arr0 = arrInt.get(0); // Unboxing
```

VIII. Helpful Methods Of Wrapper Class

As above, I said that *Wrapper* classes help create primitive data types that only contain data, becoming an object with more useful methods. Here I will list their useful methods for your reference.

1. *parseXxx ()*

The parameter passed to this *static* method is a string, the result is a primitive value corresponding to the string passed.

```
int i = Integer.parseInt("10");
float f = Float.parseFloat("4.5");
boolean b = Boolean.parseBoolean("true");
```

```
System.out.println(i); System.out.println(f); System.out.println(b);
```

You can also guess the output of the console printout of the code above, right?

2. *toString ()*

Unlike the *toString ()* a *static* method , it has to that *Wrapper* class , and the

result is a string corresponding to the value passed. This example experiments with *Integer* class , other *Wrapper* classes will be similar.

of the Object class , this *toString ()* of *Wrapper* classes is a value passed as the primitive data type corresponding

```
String sI = Integer.toString(10); System.out.println(sI);
```

The output to the console is the string "10" .

3. *xxxValue ()*

You are familiar with this method over and over. Specifically, methods of this type help to convert a value of a *Wrapper* class to the primitive data type (unboxing). Sometimes this method also helps to convert the data type as if you *explicitly cast* a value. You can understand more easily by looking at the following example.

```
Double d = 50.5; int i = d.intValue(); byte b = d.byteValue();
```

```
System.out.println(d); System.out.println(i); System.out.println(b);
```

The output to the console is 50.5 , 50, and 50 respectively .

4. *compareTo ()*

If you remember, in the string lesson we also covered the *compareTo () method* used to compare string values against each other. So for the *Wrapper* classes that we are familiar with this, its functionality is still fully applied. That is, this method will be used to compare two values of two *Wrapper* classes (of the same data type) with each other.

Specifically, with you calling *lopWrapper1.compareTo (lopWrapper2)* , the result will be as follows.

- If the method's result returns a negative number , then *lopWrapper1* will have a smaller value than *lopWrapper2* .
- If the method's result returns 0 , then *lopWrapper1* will be equal to *lopWrapper2* .
- If the method's result returns a positive number , *lopWrapper1* will have a greater value than *lopWrapper2* .

You can see the following example for better understanding.

```
Integer i = 50;
```

```
Integer i1 = Integer.parseInt("50"); Integer i2 = Integer.valueOf(52); Integer i3 =
```

30;

```
System.out.println("CompareTo i & i1: " + i.compareTo(i1));
System.out.println("CompareTo i & i2: " + i.compareTo(i2));
System.out.println("CompareTo i & i3: " + i.compareTo(i3));
```

The output to the console will be.

```
CompareTo i & i1: 0
CompareTo i & i2: -1 CompareTo i & i3: 1
```

5. *compare ()*

This method has the same usage and usage as *compareTo ()* above. The difference is that this is the *static* method of each *Wrapper* class , so you can call it directly from the class. At the same time, the parameter passed are two values of the two *Wrapper* classes . The return result of *compare ()* also has one of three values (*negative* , 0 , *positive*) as with *compareTo ()* above.

```
Integer i1 = Integer.parseInt("50"); Integer i2 = Integer.valueOf(52);
System.out.println("Compare i1 & i2: " + Integer.compare(i1, i2));
Float f1 = new Float("20.25f"); Float f2 = new Float("2.43f");
System.out.println("Compare f1 & f2: " + Float.compare(f1,f2));
```

The output to the console is.

```
first Compare i1 & i2: -1
2 Compare f1 & f2: 1
```

6. *equals ()*

Similar to *equals ()* when comparing strings. This method will compare the values of *Wrapper* classes and return a *boolean* type , where *true* is equal and *false* is different. Like the following example.

```
Integer i1 = Integer.parseInt("50"); Integer i2 = Integer.valueOf(50);
System.out.println("Compare i1 & i2: " + i1.equals(i2));
Float f1 = new Float("20.25f"); Float f2 = new Float("2.43f");
System.out.println("Compare f1 & f2: " + f1.equals(f2)); The output to the
console is.
```

Compare i1 & i2: true

Compare f1 & f2: false

- Exercise Number 1

Try executing the following code, and try to find out why the data printed to the console is like that.

```
int i = Integer.parseInt("10");
float f = Float.parseFloat("4.5a");
System.out.println(i);
System.out.println(f);
```

- Exercise Number 2

Please indicate the output of the console printout of the following line of code.

```
int i = Integer.parseInt("10.5");
System.out.println(i);
```

- Exercise Number 3

In addition to the useful methods of the *Wrapper* classes that I have listed above, please find out about other methods for yourself, they are as useful as what I mentioned. Such as.

- Static methods of *Integer* , *Long* , *Float* ,and *Double* classes : *toHexString()* , *toOctalString ()* , *toBinaryString ()* , *max ()* , *min ()* ,
- Static methods of *Character* class : *isLowerCase()* , *isUpperCase()* , *isDigit()* , *toLowerCase ()* , *toUpperCase ()* , ...

So through this's lesson, you have learned how primitive data types in Java were "surrounded" by *Wrapper* classes , to make them more diverse and convenient. is not.

Lesson 34: Exception (Part 1)

Before starting to go into detail in the lesson, I want you to know that no one is perfect in life. When people get old enough, they start making mistakes. The important thing in this life is that you don't always avoid making mistakes (because, frankly, no one wants to get in the way), but learn how to make mistakes. how to overcome and overcome mistakes.

Oh why this's lesson is so philosophical. Actually, I led a little bit for fun. The idea of this's lesson will be almost the same philosophy. That is, we will consider an ERROR aspect. ERROR here is ERRORS that occur in the code we write, seriously and in depth.

Going back to the philosophy a little bit, that once our application is large

enough (in terms of both lines of code and features), then it is certainly difficult to control stability of application logic. And when there will be an ERROR, a slight ERR will cause the system to issue some strange messages to the user, causing them to panic, the more severe error will cause the application to terminate unexpectedly. abruptly. And like the philosophy above, you do not expect ERRORS to happen, but you should understand and distinguish between the types of ERRORS in Java, to be able to provide the application with a script powerful enough to overcome ERRORS without terminate abruptly, at least report a clear piece of ERROR status, rather than letting them fiddle with the functionality your application has lost control of.

ERROR that we are talking about is referred to as *Exception* in Java. And because of its importance, the issue will be talked about in many different sections. Invite you to familiarize yourself with the first episode in the series about this *Exception*.

I. Exception Concept

Exceptions can be translated into Vietnamese as *Exceptions* or *Exceptions*.

Why are we talking *Error* and not *Error*? The *exception* actually includes *Error* in it. *Exception* is a concept used to refer to a phenomenon when the logic of the application is affected (in a bad way) by a certain problem (either an error or not). *When an exception occurs, it causes the program to deviate from the standard thread that we have programmed*, leading to the application's inability to process its logic, or possible abrupt discontinuation (this state has can be called "*death*", "*sudden death*", or "*crash*").

This *exception* happens in an unexpected way. And even if you have prepared the situations carefully, the flow deviation remains latent. As the following examples show you more clearly where the danger is actually coming from.

An *exception* can happen when the user enters incorrect data into the application. By accident or on purpose. As in *Exercise 1 of the previous exercise*, if you want a real number for the program, the user enters a!?!?

- Or when the application finds and reads a file, it cannot see it. Maybe that file has been deleted before.
- Or the system runs out of memory while the application is running that makes

the application unable to execute its logic.

And many other situations too. In short as I said, *Exception* can be generated from certain errors, such as programmer's code error, or user input error. And *exceptions are* also born from situations that are not errors, such as running out of memory, dropping the network connection, such as accidental user intervention, ...

You have to understand why it is called *Exception* , right? To understand more clearly, we see what types of *exceptions* we classify .

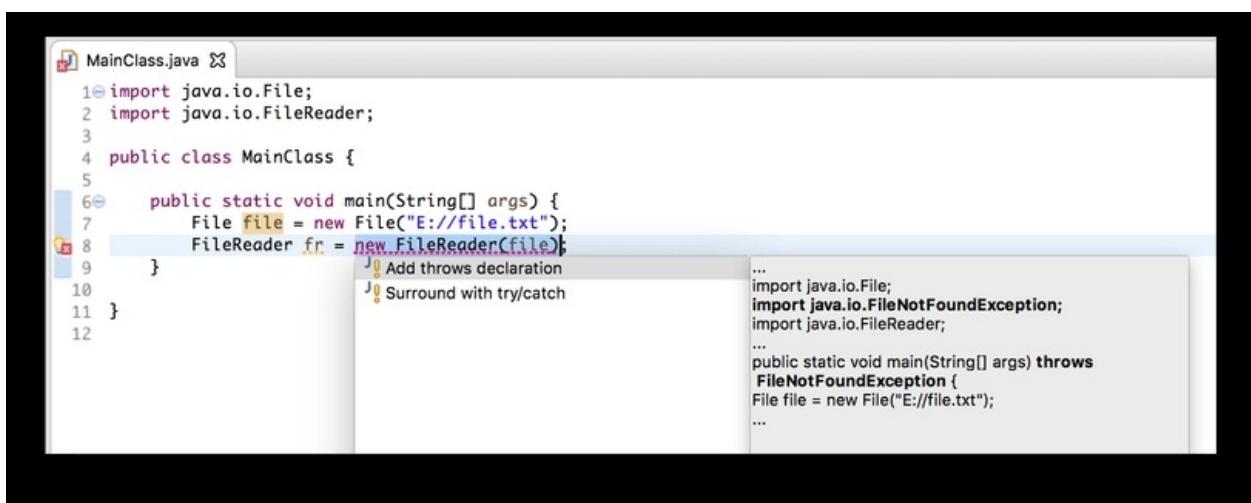
II. Exception Classification

To understand and easily work with *Exception* , Java divided it into 3 categories as follows.

1. Checked Exception

These are *exceptions that can* not pass the "gate" of the compiler. That is, you will get error messages from the compiler when it discovers that you are coding certain lines of code that has the possibility of an *exception* . And because the compiler has already discovered the *exception* for you, the name is *Checked* , which means "*moderated*" .

If you want to try out the compiler, in particular *Eclipse* , try typing the following commands into the *main* method . You do not need to know what the following lines of code actually say, but just try typing, make sure to *import java.io.File* and *java.io.FileReader* .



The screenshot shows an Eclipse IDE window with a Java file named "MainClass.java". The code is as follows:

```
1 import java.io.File;
2 import java.io.FileReader;
3
4 public class MainClass {
5
6     public static void main(String[] args) {
7         File file = new File("E://file.txt");
8         FileReader fr = new FileReader(file);
9     }
10
11 }
```

A code completion tooltip is displayed at the cursor position on the line "FileReader fr = new FileReader(file);". The tooltip contains two options: "Add throws declaration" and "Surround with try/catch". Below the options, the completed code is shown:

```
...
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
...
public static void main(String[] args) throws
FileNotFoundException {
File file = new File("E://file.txt");
...
}
```

See, you have seen the compiler detect and error. But unlike the usual error, if it is an error, like you mistyped `System.out` to `Ssystem.out`, for example, you must fix the error right away, but with *exceptions* , we don't fix anything, but Find a way to "catch" bugs and break the logic of the application in a different direction, which we will take a closer look at later.

2. *Unchecked Exception*

This *exception* is quite dangerous, when the compiler cannot check it to help you. And your application when it reaches the user, when the user is manipulating it... boom! Application "*sudden death*" .

Because this *exception* occurs while the application is being executed, it is also called the *Runtime Exception* .

Normally, when an application dies of this kind, the system still has a log that tells us what the error was. You can try to create this *exception* when trying to compile the following code.

```
1 int num[] = {1, 2, 3, 4};  
2 System.out.println(num[5]);
```

The code seems harmless (because there is no error when coding). But the system will not be able to find the *5th* array element if the application is executed, because our array only declares *4* elements. And so the application dies without knowing what else to do. You can try to execute and view the log in to the console with the above code.

3. *Error*

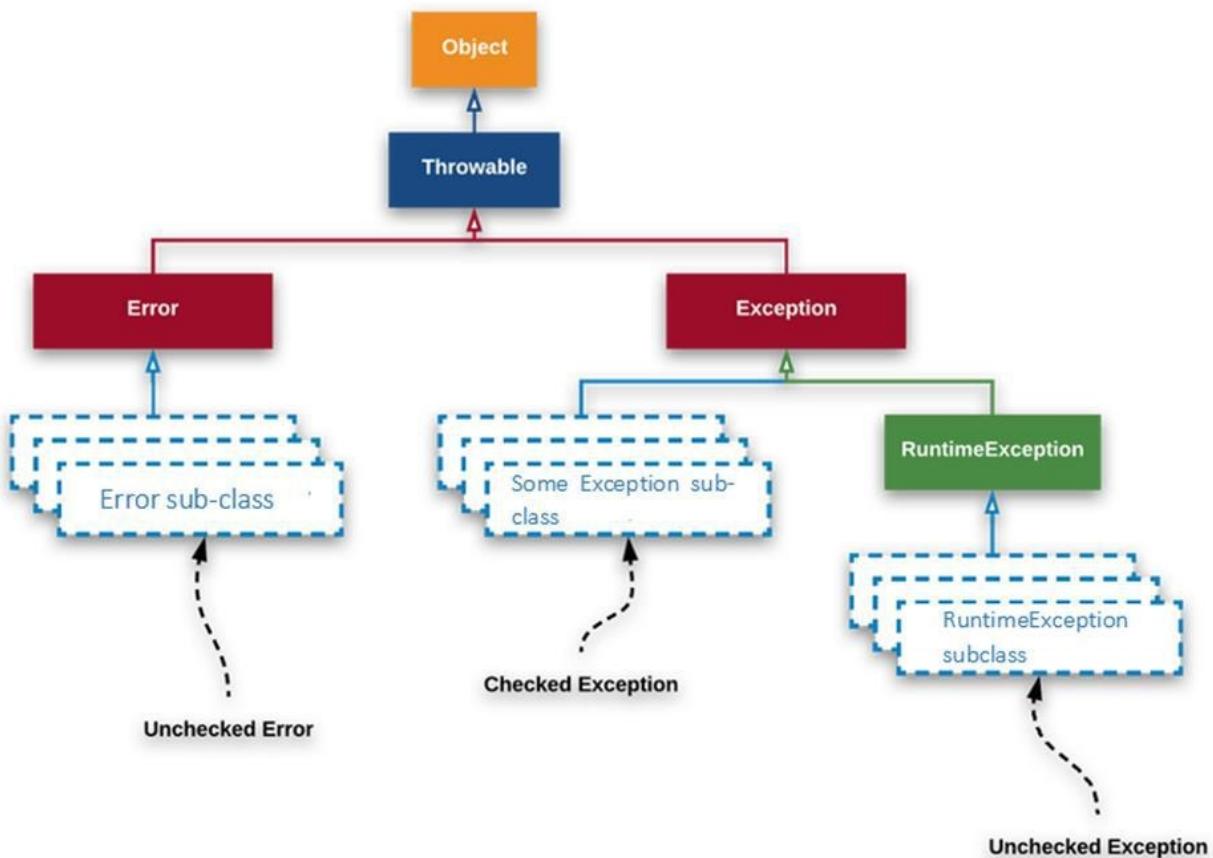
Error here is also *Exception* but slightly different from *Exception* . With the two *Exceptions* that I mentioned above, actually not an error, because you should not avoid it, but should face and "break" the application flow so that when encountering this *Exception* , the application will know how to respond properly. right. Next lesson we will practice breaking this thread.

Also *Error* really a thing could not do anything else. It is beyond our disposal. Even if you know in advance, an *error* may occur. When the application *crashes* , usually we can just re-code the place for it differently and then update a new version of the application, but do nothing. For example, when the system

memory has been exhausted, or when you are using a library, and unfortunately one day a class or method of that library is no longer found.

III. Hierarchical Tree Of Exception Classes In Java

The *exceptions* mentioned above are the classes in Java. When the application deviates from the thread, the system will also assign that error specifically to a certain *Exception* class to manage. And because *Exceptions* are divided into 3 categories like above, you can see the classification of these *Exception* classes also divided into 3 main groups.



If you do not understand anything about *Exception*, just take a look. Lesson after we started "catch" the *exception* based on the inner layers of hierarchy in the diagram above. And we can also create our own *exceptions* according to the requirements of the application.

As far as the hierarchy is concerned, the top class relevant to all of this's lessons

is the *Throwable* class . We will get familiar with *Throwable* and the useful methods of this superclass in the next lesson, as we already know how to catch the *exception* .

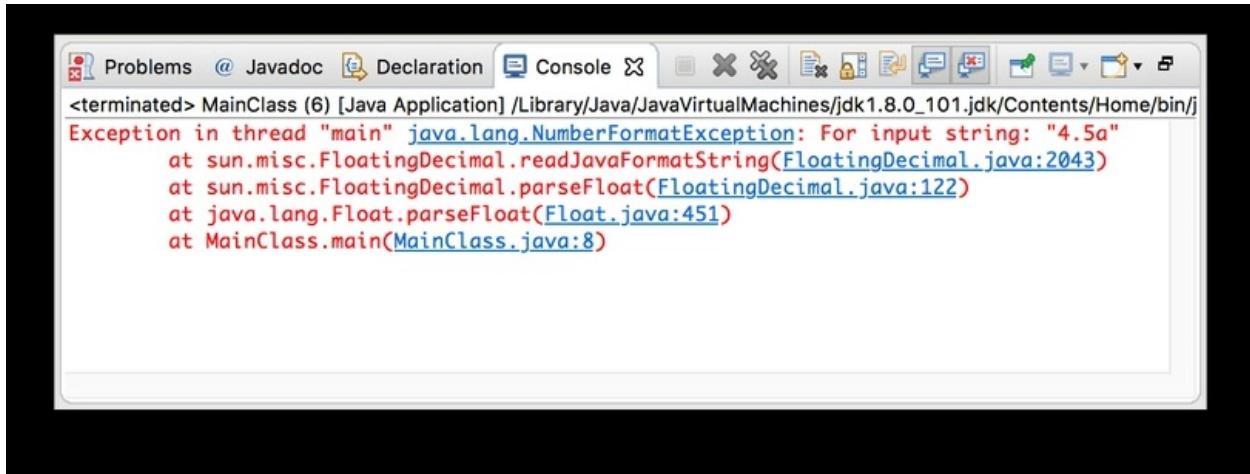
The two main classes of *Throwable* are *Exception* and *Error* . In *Exception* there are two types of subclasses. One type is *RuntimeException* and its children , they are *Unchecked Exceptions* as I mentioned above, *Exceptions* in this branch will not be censored by the compiler, and so it is potentially risky. applications are running in the real world. The remaining children of *Exception* outside *RuntimeException* that I just talked about above is *c evil other exception* , they are all *Checked Exception* , if you use the code can "touch" to the *exception* in this branch will be the system error, as examples that you have seen on the other. The *Exceptions* on the *Error* branch will not be reported by the system, it will find hidden risk of application dying while running that we cannot anticipate, as we mentioned above.

- Exercise Number 1

Try to code the lines of *Exercise 1 of the previous lesson* , but this time pay close attention to the *Exception* that the console announced.

```
int i = Integer.parseInt("10");
float f = Float.parseFloat("4.5a");
```

```
System.out.println(i); System.out.println(f);
```



Try to find out what kind of *Exceptions* the *NumberFormatException* that the console displays is: *Checked Exception* , *Unchecked Exception* , or *Error* ?

Hint: you can use *Ctrl + left click* (or *Command + left click* with Mac) on any

layer in Eclipse to be led to the declaration of that class in the system, you will understand more about the relationship of that class to other classes.

- Exercise Number 2

Similarly, you try to code the following line and tell what its *exception* is, and what kind of *exception* is it.

```
int value = 10 / 0;  
System.out.println(value);
```

- Exercise Number 3

Try to figure out what exception this code below going to make String s = null;
System.out.println(s.length());

So with this's lesson, we are familiar with what is *Exception* in Java. We will have a few more lessons to be able to talk about all the interesting knowledge about this *exception* , please keep following it.

Lesson 35: Exception (Part 2)

In the previous lesson, you were familiar with the concept of *Exception* , and tried to create a few *exceptions* to "watch and play" together . However, the main thing that *Exception* was born is not that we can only stand and look helplessly like that. We can completely deal with *Exceptions* , by capturing them, blocking them from making the program crash, and doing what we want, like in the previous lesson, we used the word "cracking" .

Thus, the second part of the *Exception* series this we talk about the techniques to catch these *Exceptions* .

You can review the *exception* knowledge in this *first part* .

I. Be Familiar With Try Catch

As the first part of the lesson has an introduction, this we are looking to *catch an exception* , or some documents called "bug trap" .

And of course, whatever it is, if we want us to do something, we have to give us the tools to do it. The tool that traps the *Exception* that we are talking about is called *Try Catch* . *Try catch* is a tool that helps you wrap around the code that has the possibility of an *exception* . Or the code that has been error reported by the system (which are the *Checked Exceptions*). The code we talked about will

be wrapped in the *try* block. Then, if that *exception* happens in that *try* block, the system will quickly "*break*" the logic flow of the application, moving to executing the code inside the *catch* block.

Based on the above explanation, please see the syntax for *try catch* as follows.

```
try {  
// The lines of code can cause Exception  
} catch (ExceptionClass e) {  
// If an exception occurs, these lines of code will be called  
}
```

To make it easier to understand, please *try to catch a test* for a possible error case that in the previous lesson you already know through the following exercise.

1. Practice Building Try Catch

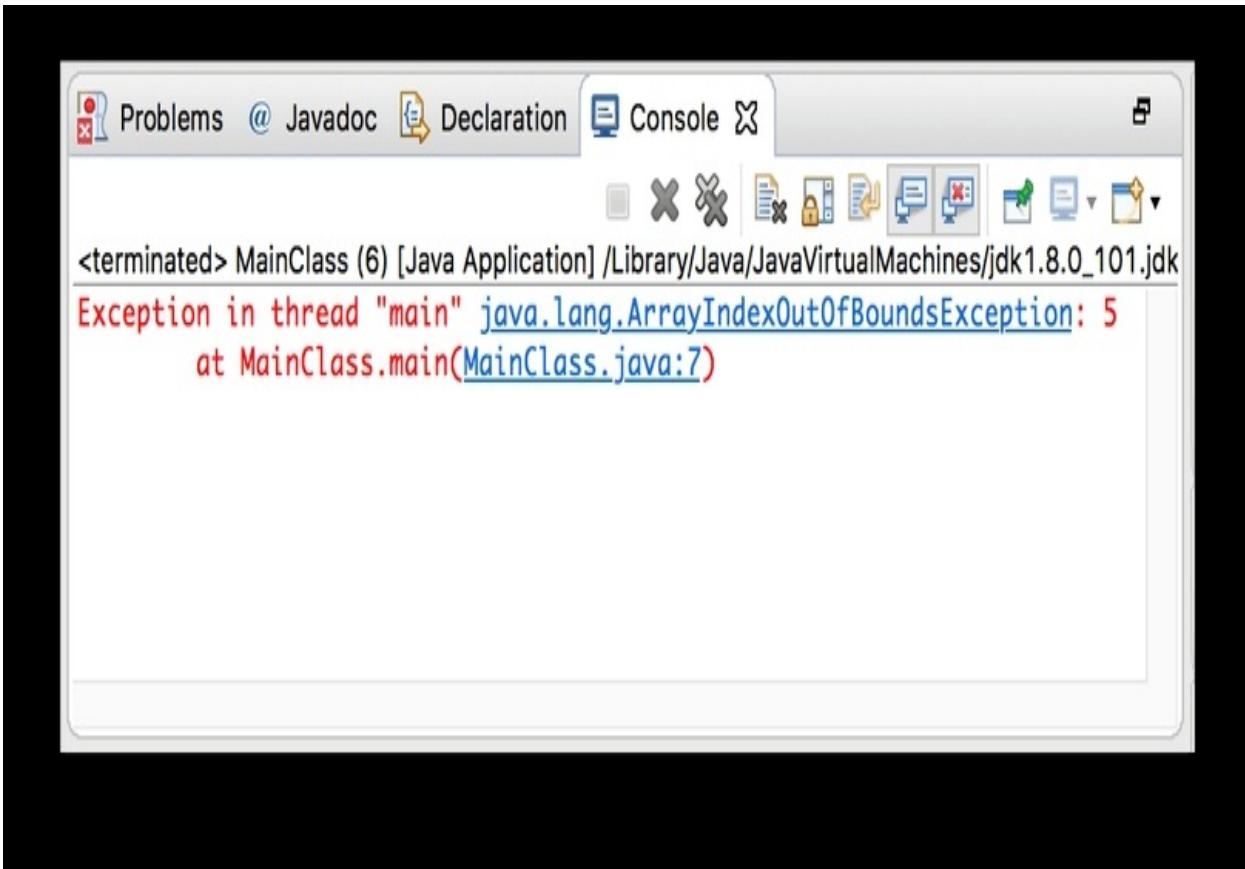
We have a look at the types of code that can cause *exceptions*, you can review them in *this section*. And here I will write them down.

```
int num[] = {1, 2, 3, 4};  
System.out.println(num[5]);
```

You already know that this code falls into an *Unchecked Exception*, which means the compiler will not recognize and check for you whether an *exception* has occurred or not.

But if the compiler doesn't do anything, let's not cross our arms. To make sure our application is "*healthy*", you should anticipate *exceptions* and include "*potentially pathogenic*" code in the *try catch* block .

If you stubbornly execute the application, you will get an error message like this. Please pay attention to the *Exception*, we will *catch* them below.



Let's wrap the fragment that can cause an error with the *try catch* block like this.

```
try {  
    int num[] = {1, 2, 3, 4};  
    System.out.println(num[5]);  
  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("Cannot print because the array element  
    was not found as expected.");  
  
}
```

At this point, if you re-execute the application, the user will be easier to understand with a more "*human*" error message as follows.

Cannot print because the array element was not found as expected

2. Practice Verifying How Try Catch Works

With this exercise you will better understand the essence of "*trapping*" errors and how the execution of *catch* lines occurs immediately when the "*trap*" has caught the error.

You see the code with the full *try catch* as follows. Guess what the output to the

console will look like.

```
try {  
    System.out.println("The program is doing some divides:");  
    int result = 10 / 2;  
    System.out.println("10 divided by 2 equals " + result);  
    result = 10 / 1;  
    System.out.println("10 divided by 2 equals " + result);  
    result = 10 / 0;  
    System.out.println("10 divided by 0 equals " + result); } catch  
(ArithmaticException e) {  
    System.out.println("There is one divide doesn't work ");}  
And this is the result.  
The program is doing some divides:  
10 divided by 2 equals 5  
10 divided by 2 equals 5  
There is one divide doesn't work
```

Oh. The line prints consolde *System.out.println ("10 divided by 0 equals" + result)*; where is it? In fact, as soon as the system detects that an *exception* in the *try* block occurs, the code below of that *try* block will not be executed anymore, but makes room for the blocks in the *catch that* execute next. As a little fun illustration is when the code in line 10 is excuted, it cause exception and jump to line 13.

II. Some Rules With Try Catch

At this point, you already know how to wrap the lines of code that are capable of causing *Exception* by *try catch* . However, I know that for you who are just starting to get acquainted with *try catch*, there will be many questions, I would like to list them in the form of the following rules.

1. *The Stream Don't Always Get In The Catch*

As you are familiar with, the statements inside the *try* block do not always cause *exceptions* , and so not always the statements inside the *catch* block are executed, if the logic in the *try* is executed. safe. As in the above example, if you do not divide 10 by 0, there will be no statement saying " *There is an unworkable division*" on the screen.

2. But If You Defined Try, You Must Define Catch

However, not always the code in the *try* causes an *exception*. But if there is a definition *try*, then you must have a *catch* definition attached. That is a guarantee. That you must always specify a fallback logic in case an *exception* occurs.

3. Can Catch With Exception Class?

You don't have to catch with specific subclasses like *ArrayIndexOutOfBoundsException* or *ArithmaticException* as in the examples above. You can also use the *exception* superclass. Then the statement will be *try {...} catch (Exception e) {...}*. Because you don't always remember the subclasses of *Exception*! But be careful! Let's learn to use specific *child Exceptions*, don't be too lazy to use the *Exception* class, even the *Throwable* class instead.

For what, of course at first your code will be easier to manage, and releasing bugs to the user will be more specific, which will be clear when you get used to catching multiple bugs at the same time. as below.

III. Try With Many Catch

There are times when we cram too much code into the *try* block, making it *possible to cause more than one Exception*. Then we can completely apply the following syntax to be able to catch many *Exceptions* in the same *try*.

```
try { // The lines of code can cause Exception
} catch (ExceptionClass1 e1) {
// Catch the EceptionClass1
} catch (ExceptionClass2 e2) {
// Catch the EceptionClass2
} catch (ExceptionClass3 e3) {
// Catch the EceptionClass3
}
```

According to the above syntax, you will still wrap the statements that have the ability to cause an *exception* in a *try* block. Then each *catch* will be a separate *Exception*.

Then, for any line that causes an *exception*, the system will in turn look for

catch blocks below *try* until it finds a corresponding *exception* .

For Java 7 , you can write syntax *try* with much *catch* as follows. The *exception* in this notation is in the same *catch* alone but separated by the character " | ".

```
try { // The lines of code can cause Exception
} catch (ExceptionClass1 | ExceptionClass2 | ExceptionClass3 e) { // Catch
ExceptionClass1, EceptionClass2 and EceptionClass3
}
```

We will get familiar with building many *catches* like this in the next exercise.

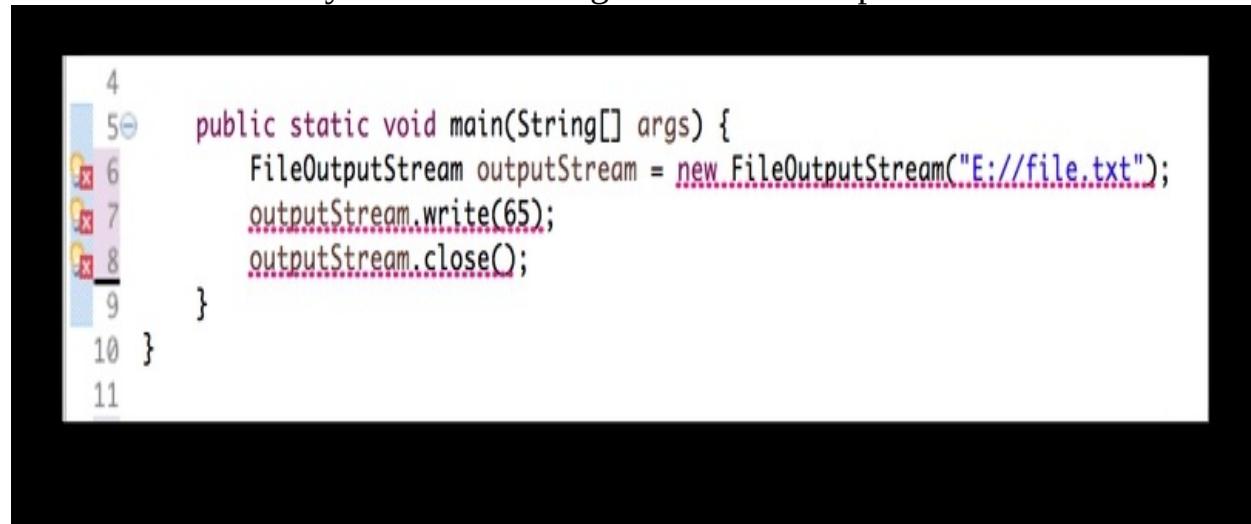
1. Practice Building Try With Many Catch

In this exercise, you will be familiar with *Checked Exceptions* , and get acquainted with the fastest way to add *try catch* blocks (or *try* with multiple *catches*) in Eclipse.

First, code the following line into a method.

```
FileOutputStream outputStream;
outputStream = new FileOutputStream("E://file.txt"); outputStream.write(65);
outputStream.close();
```

You will immediately see error messages from the compiler.

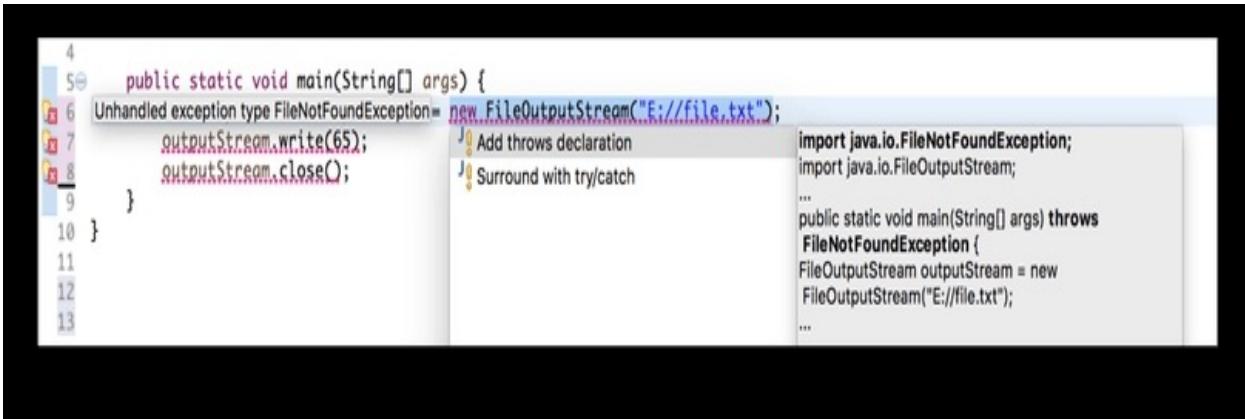


A screenshot of an IDE showing a Java code editor. The code is as follows:

```
4
5  public static void main(String[] args) {
6      FileOutputStream outputStream = new FileOutputStream("E://file.txt");
7      outputStream.write(65);
8      outputStream.close();
9  }
10 }
11 }
```

The code contains several errors, indicated by red X icons in the margin on the left side of the editor. Lines 5, 6, 7, and 8 each have a red X icon, while lines 5, 6, 7, and 9 each have a red X icon.

This is not necessarily an error. Because when you click on the bulb icon on the left, you will see a fix as follows.



Obviously this is a *Checked Exception* , and the system is suggesting you with two options.

- *Add throws declaration* : this option allows you not to create any *try catch* block , but continue to "throw to someone" *try catch* helps. This technique will be discussed in the next lesson.
- *Surround with try / catch* : this is the option we want, with choosing this option, the system will wrap your code with a *try*

catch block with *Exception* named *FileNotFoundException* .

You can manually type in *try catch* as suggested. But if you choose the second option, the system will *try catch* automatically for you, but this automatic is quite "clumsy" , you should just fix the code to look nice as follows.

```
try {
FileOutputStream outputStream;
outputStream = new FileOutputStream("E://file.txt"); outputStream.write(65);
outputStream.close();

} catch (FileNotFoundException e) {
// TODO Auto-generated catch block
e.printStackTrace();

}
```

And you can see, though did *try catch* with *FileNotFoundException* , the system still is error. Let's see.

The screenshot shows a Java code editor with the following code:

```
5 public static void main(String[] args) {
6     try {
7         FileOutputStream outputStream;
8         outputStream = new FileOutputStream("E://file.txt");
9         outputStream.write(65);
10    } ca J! Add catch clause to surrounding try
11    J! Add throws declaration
12    J! Add exception to existing catch clause
13    J! Surround with try/catch
14 }
15 }
16 }
17 }
18 }
```

A code completion dropdown menu is open at the cursor position (line 10, after the opening brace). The menu contains the following options:

- ... (ellipsis)
- import java.io.FileNotFoundException;
- import java.io.FileOutputStream;
- import java.io.IOException;
- ...
- public static void main(String[] args) throws IOException {
- try {
- ...

It turns out that the system needs you to *catch* another *exception* named *IOException*. At that time there could be more options. As shown above, they are included.

- *Add throws declaration* : like I said above.
 - *Add catch clause to surrounding try* : add one more *catch , try catch catch* form like the first syntax in this section.
 - *Add exception to existing catch clause* : add an *exception* in the existing *catch* . The *try catch* form (*ExceptionClass1 | ExceptionClass2*) is like the second syntax in this section.
 - *Surround with try / catch* : the cage one more *try catch* to *try catch* the present, this way your code less applicable because look complicated.
- At this point you can choose the second way the code looks like this.

```
try {
    FileOutputStream outputStream;
    outputStream = new FileOutputStream("E://file.txt");
    outputStream.write(65);
    outputStream.close();

} catch (FileNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();

} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
```

}

Or choose the third way to generate this code.

```
try {  
    FileOutputStream outputStream;  
    outputStream = new FileOutputStream("E://file.txt"); outputStream.write(65);  
    outputStream.close();  
  
} catch (FileNotFoundException | IOException e) { // TODO Auto-generated  
catch block  
e.printStackTrace();  
  
}
```

- Exercise Number 1

Let's say I have the following program. The program will generate *10* random integers and store in an array of *10* elements. The program will let the user enter an index of the array and then output the value of that array to the console.

```
// Creates an array of 10 random integers  
int randomIntNumbers[] = new int[10];  
Random rand = new Random();  
for(int i = 0; i < 10; i++) {  
  
    randomIntNumbers[i] = rand.nextInt(100); }  
  
// Let the user enter an integer and print to the screen  
// corresponding array element  
Scanner input = new Scanner(System.in);  
System.out.println("What array element do you want to print out?"); int index =  
input.nextInt();  
System.out.println("OK, the array element in" + index + " has value "  
  
//+ randomIntNumbers[index]);
```

Of course, this program has the potential to cause *exceptions* at runtime. Please try to clear the corresponding *try catch* so that when an *exception* occurs, the program can promptly notify the error to the user.

And here is the answer code, you should code and "*break*" the program yourself

before clicking on this answer.

```
// Create an array of 10 random integers int randomIntNumbers [] = new int [10];
Random rand = new Random ();
for (int i = 0; i < 10; i++) {

    randomIntNumbers [i] = rand.nextInt (100); }

try {
    // Let the user enter an integer and print to the screen
    // corresponding array element
    Scanner input = new Scanner (System.in);
    System.out.println ("What array element do you want to print?"); int index =
    input.nextInt ();
    System.out.println ("OK, the array element" + index + "has value"

    + randomIntNumbers [index]);
} catch (InputMismatchException e) {

    System.out.println ("Invalid array element, please enter an integer!"); } catch
    (ArrayIndexOutOfBoundsException e) {
    System.out.println ("Invalid array element, please enter a value between 0 and
    9!");
}
• Exercise Number 2
```

I have the following code that has been "*careful*" *try catch* , you see if the *try catch* can still cause any errors.

```
Scanner input = new Scanner(System.in);
Integer intNumber = null;

try {
    System.out.println("Please enter an integer: "); String strNumber =
    input.nextLine();
    intNumber = new Integer(strNumber);

} catch (NumberFormatException e) {
    System.out.println("Please enter an integer ");
    System.out.println("Convert to Hexa: " + Integer.toHexString(intNumber));
```

And I discovered it was not good. Because if there is an *Exception* happening

with the section calling the user to enter a number, then the *intNumber* has not been initialized (with a *null* value), and another *exception* will occur for the final code. I revised this code as follows.

```
try {  
    Scanner input = new Scanner (System.in); Integer intNumber = null;  
  
    System.out.println ("Please enter an integer:"); String strNumber =  
    input.nextLine ();  
    intNumber = new Integer (strNumber);  
  
    System.out.println ("Convert to Hexa:" + Integer.toHexString (intNumber)); }  
catch (NumberFormatException e) {  
    System.out.println ("Please enter an integer");  
}  
You have just finished watching part 2 of the series about Exception . Through  
this lesson, we got used to catching errors and breaking the flow of the  
application through the try catch tool . Try catch will also have "variation" again  
and I will spend the next part to continue.
```

Lesson 36: Exception (Part 3)

Through two lessons about Exception, you have somewhat more peace of mind for the "fate" of the lines of code that you have created, right. The truth is that with the good use of *try catch* that the previous two lessons I have talked about very carefully, it can be said that your application will become very powerful, safe, and also quite smart when it can be notified. time error cases for the user.

However, if we talk about *Exception* , we must say until. Do not be half-hearted and miss out on the great functions that this tool offers. This we continue to delve into *Exception* when talking about *try catch with finally* , *try catch with resource* , and *useful methods of Exception class* .

I. Try Catch With Finally

I repeat a bit from the previous lessons, that when you are familiar with how to catch an *exception* through a *try catch* , *try* helps to wrap the code that is likely to cause an error, while the *catch* will help handle the "consequences" when but the code above was officially faulty.

However that is not enough. In Java, sometimes we use up some system resources. This resource is the resource that can be shared between applications. It could be as simple as a file. And because these are shared resources, if there is a situation when your application is opening a resource, but unfortunately causes an exception in this process, then the application will report an error. , and... done... carelessly does not close the resource. In this case, the system thinks that the resource is still being managed by your application, but in fact your application is dead for a long time.

Thereby you can see that, in programming you should not let the application hold any resource for you, you should close it after using it. You will understand this when familiar with Input / Output knowledge in the next lesson. From the above situation, *finally* was born. *Finally* will help us clean up the remnants from the *try catch* left behind, in this's lesson its main task is to close the resources when an *exception* occurs. First of all, please see the following syntax.

```
try { // The lines of code can cause Exception  
} catch (ExceptionClass1 e1) {  
// Catch the EceptionClass1  
} catch (ExceptionClass2 e2) { // Catch the EceptionClass2  
} catch (ExceptionClass3 e3) { // Catch the EceptionClass3  
} finally {  
// Free up resources in use  
}
```

Through the above syntax, you can see that *finally* will accompany a *try catch* . If there is no *try catch*, *there* will be no *finally* . But no matter how many *catches* *try* comes with , just having a *finally* to clean up what you have used is enough.

Below are a few points related to *finally* , I hope these items will help you have a clear view of this *finally* .

1. Just There is Try - Finally (Without Catch) Also Okay

You can understand that *finally* in this case is used to clean up something without necessarily having an *exception* happen. Because the need to use *finally* is very little, the example below will not be very practical, mainly to help you understand.

```
try {
```

```
System.out.println("The application is doing math:");
int result = 10 / 2;
System.out.println("10 divided by 2 equal: " + result);

} finally {
System.out.println("This is the end of program");
}
Surely you also know the results. That all 3 lines printed to the console above are done smoothly.  
The application is doing math:  
10 divided by 2 equal: 5  
This is the end of program
```

2. You May Not Need To Finally, But If There Is Finally in The Code, Finally Will Always Be Called Last

For example, *try catch* does not need *finally* , you already know through *the previous lesson* . But if you have a *finally* declaration for *try catch* , the statements in the *finally* block will be executed eventually, after the application has executed the statements that do not cause an *exception* in *try* , then the statements in the *catch* block. respectively, if any, finally reach *finally* .

```
try {
System.out.println("The application is doing math:");
int result = 10 / 0;
System.out.println("10 divided by 0 equal: " + result);

} catch (ArithmaticException e) {

System.out.println("Can't perform the code");
} finally {
System.out.println("This is the end of program ");
}
```

With these lines of code, the console will print as follows. And maybe I don't need much explanation.

```
The application is doing math
Can't perform the code
This is the end of program
```

3. In Finally Can Still Have Try Catch In It

If in *try* or *catch* we can nested more *try catch* . Then *finally* allows the same. Although it looks complicated and a bit unsafe, but life is not perfect at all, it is

common to have errors in the process of catching mistakes.

You will be familiar with *finally* and also *try catch* inside *finally* through the practice below.

4. Practice Building Try Catch With Finally

This exercise helps us get used to the more practical use of *try catch* and *finally*. We will inherit the code from *the previous exercise*.

Because of the exercise the day before, we have just learned how to catch the *exception* effectively. In fact, you are using *FileOutputStream* to open a file from the system and manipulate that file. This file is a resource that can be shared by other programs. Like I said, if you use this file and if something happens and you don't release ownership, then you are "*selfish*" to just keep the file for yourself, other applications. can read this file.

That is why we need to construct *finally* to release this resource as follows. First, we need to code the lines that cause the *Checked Exception* as follows.

```
FileOutputStream outputStream = null;  
outputStream = new FileOutputStream("E://file.txt"); outputStream.write(65);  
outputStream.close();
```

And as with the practice in the previous lesson, you already know the easiest way for the system to manually add *try catches* for lines that are reporting errors. After applying the hints from Eclipse (see previous lesson), our code looks pretty good like this.

```
FileOutputStream outputStream = null;  
try {  
    outputStream = new FileOutputStream("E://file.txt");  
    outputStream.write(65);  
    outputStream.close();  
} catch (FileNotFoundException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
} catch (IOException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}
```

What is the problem with the above code? You know that a file named *file.txt* is treated as a shared resource. This means that while you open this file to write to the value 65 , at the same time, there may be another program trying to manipulate this file. And the problem happens is, as if in the process of writing to the file, there are any errors that occur, causing the program to be broken into the lines of code inside the *catch* block below. Then the *outputStream.close () statement* will have no chance to execute. The file you manipulate will still be considered by the system to be in use. And so other applications have to wait forever without seeing your application return permission to use this file. Issues with file manipulation we will be familiar with in the following lessons. For now, make sure that the file you are using is always securely closed. Return resource usage to other programs. It's simple, let's add *finally* and the code to close this file in this block is done. As shown below.

```
6
7  public static void main(String[] args) {
8      FileOutputStream outputStream = null;
9      try {
10          outputStream = new FileOutputStream("E://file.txt");
11          outputStream.write(65);
12          outputStream.close();
13      } catch (FileNotFoundException e) {
14          // TODO Auto-generated catch block
15          e.printStackTrace();
16      } catch (IOException e) {
17          // TODO Auto-generated catch block
18          e.printStackTrace();
19      } finally {
20          outputStream.close();
21      }
22 }
```

Oh but look, the code to enter in *finally* is still under the authority of the *Checked Exception* . Very well, we should also need one more *try catch* . The final code will look like this.

```
FileOutputStream outputStream = null;
try {
    outputStream = new FileOutputStream("E://file.txt");
    outputStream.write(65);
    outputStream.close();
```

```
} catch (FileNotFoundException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
} catch (IOException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
} finally {  
    try {  
        outputStream.close();  
    } catch (IOException e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
}
```

II. Try Catch With Resource

This type of *try catch* has been around since Java 7. And this is an upgrade over *try catch with finally* that you have just become familiar with.

Specifically, through the exercise above, you have seen the importance of freeing up system resources, right. And you also saw how to close the file in *finally* block . But if we pay close attention to the above exercise, we can see that the *finally* block will be called whether or not the *FileInputStream* initialization statement executes perfectly, so closing the file in the *finally* block can cause a Other *exceptions* . And also for this reason, we have to add *try catch* inside *finally* .

But manually executing *finally* and *try catch* inside *finally* as in the above example is also bad, is that if an *exception* occurs with code in *try* and also with code in *finally* , the printed error lines will be messed up. .

That's why Java 7 has given us a new tool called *try catch with resource (Try With Resources)* . Its syntax is as follows.

```
try (// Initialize resources) { // Use the resources created  
} catch (ExceptionClass1 e1) { // Catch the EceptionClass1  
} catch (ExceptionClass2 e2) { // Catch the EceptionClass2  
} catch (ExceptionClass3 e3) { // Catch the EceptionClass3
```

}

As you can see, this *try catch* is no longer included with *finally*. Since the resources initialized inside *the try parentheses* as above can now know by themselves how to close at the end of the *try block*.

Before getting into the detailed exercise, let's go through a few main points of *try catch with resources* to help us understand better.

1. Not Any Resource Can Be Used In Try Catch With Resource

Only objects that the *FileOutputStream* in the *resource* . implement the *java.lang.AutoCloseable interface* , like

above example, can be used in a *try catch against the*

2. During Closing System Resources, If An Error Occurs There Will be No Exception Of This Process

This means that, if with the use of *try catch with finally* above, assuming when you initialize the *FileOutputStream* fails, the *Exception* of this initialization is thrown, the program is threaded into the corresponding *catch* , this. then you know all too well. But then the code inside the *finally* block executes, if closing the *FileOutputStream* via the *outputStream.close () statement* still causes an error, the *exception* of this closure is released again, and the program is threaded into the side *catch*. in this *finally* .

But with *try catch with resources* , it is different. If the *FileOutputStream* initialization fails, only the *exception* of the *try will* be thrown and the program is threaded into the corresponding *catch* . Closing the *FileOutputStream* if there is an error or not, the program will not throw any *exceptions* . That is the basic difference between the two *try catches* that we are talking about this.

3. Multiple Resources Can Be Generated Inside Try Block

There are times when we want to secure more than one resource inside the *try block* , we just initialize these resources and separate them with (;). Details about this idea will be shown in the following exercises.

4. Practice Building Try Catch With Resource

We will modify the code of the above exercise by trying to *catch the resource* .

It's very easy and fast, you just need to enclose the resource initialization code in the parentheses of *try* , and remove the *finally* block (and all code closing resources) of the above exercise.

```
try(FileOutputStream outputStream = new FileOutputStream("E://file.txt")) {  
    outputStream.write(65);  
  
} catch (FileNotFoundException e) { // TODO Auto-generated catch block  
    e.printStackTrace();  
  
} catch (IOException e) {  
    // TODO Auto-generated catch block e.printStackTrace();  
  
}
```

5. Practice Building Try Catch With Many Resources

As promised above, this exercise will help you understand how *try catch* is with initializing more than one resource within *try* 's parentheses .

You may not understand all of the following statements, but just practice, we will talk more about these commands in the related lessons later. The following commands will read any file you specify. And since the following lines of code contain two resources that have *java.lang.AutoCloseable interface* implementation , they are *FileInputStream* and *BufferedInputStream* , we will initialize them inside *try* 's parentheses as follows. These two initialized resources will be automatically closed after the *try catch* block ends.

```
try(FileInputStream inputStream = new FileInputStream("E://file.txt"));
```

```
    BufferedInputStream bufferInputStream = new  
    BufferedInputStream(inputStream)) { int data = bufferInputStream.read();  
    while(data != -1) {
```

```
        System.out.print((char) data);  
        data = bufferInputStream.read();
```

```
    }  
} catch (FileNotFoundException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
} catch (IOException e) {
```

```
// TODO Auto-generated catch block
e.printStackTrace();
}
```

III. Some Helpful Methods Of The Exception Class

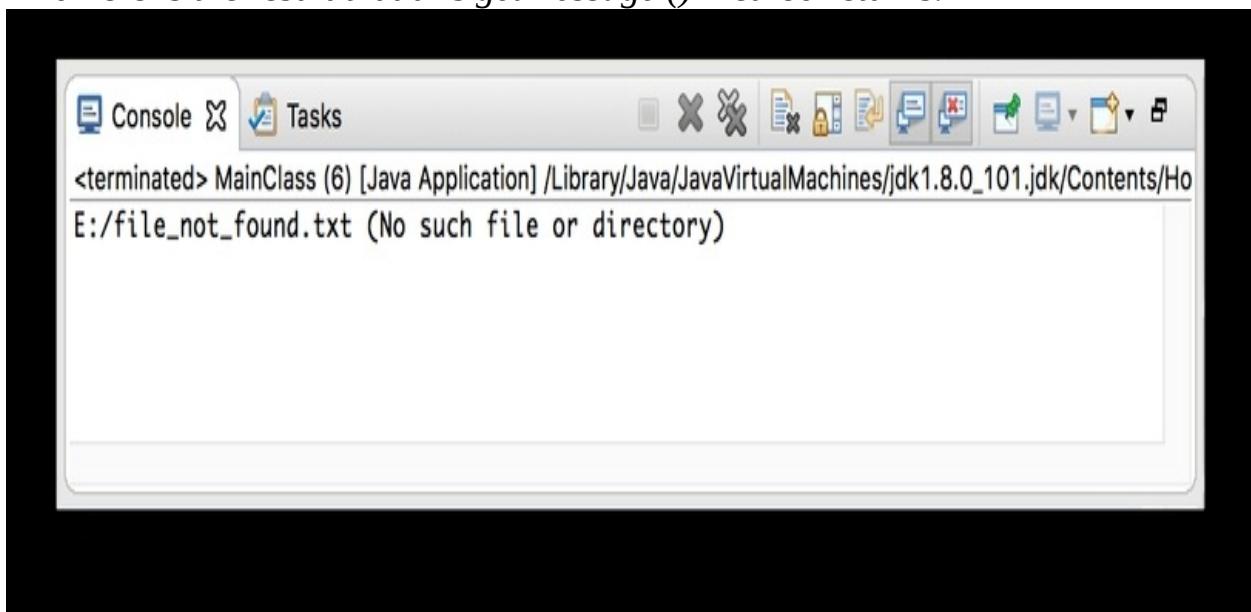
This section is my additional part in this's lesson. Although I only mentioned a few methods, but I hope you will understand more about how to use *Exception* and apply it to some specific circumstances.

1. *getMessage ()*

This method returns a String describing the *Exception* that just happened. You can experiment by calling the following lines of code. I deliberately make the lines of code cause *Exception* .

```
try(FileOutputStream outputStream = new
FileOutputStream("E://file_not_found.txt")) { outputStream.write(65);
} catch (FileNotFoundException e) {
System.out.println(e.getMessage());
} catch (IOException e) {
System.out.println(e.getMessage());
}
```

And here is the result that this *getMessage ()* method returns.

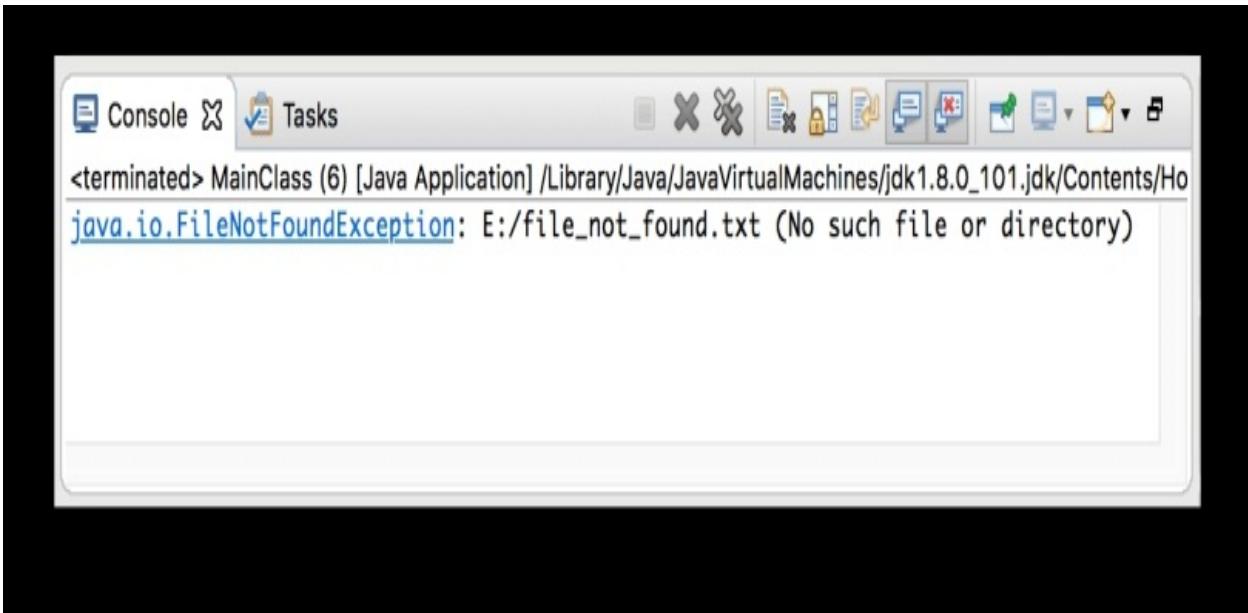


2. *toString ()*

Returns the name of the *Exception* class that is throwing the error, along with the contents of the *getMessage () method* above. You can try.

```
try(FileOutputStream outputStream = new  
FileOutputStream("E://file_not_found.txt")) { outputStream.write(65);  
} catch (FileNotFoundException e) {  
System.out.println(e.toString());  
} catch (IOException e) {  
System.out.println(e.toString());  
}
```

And this is the result.



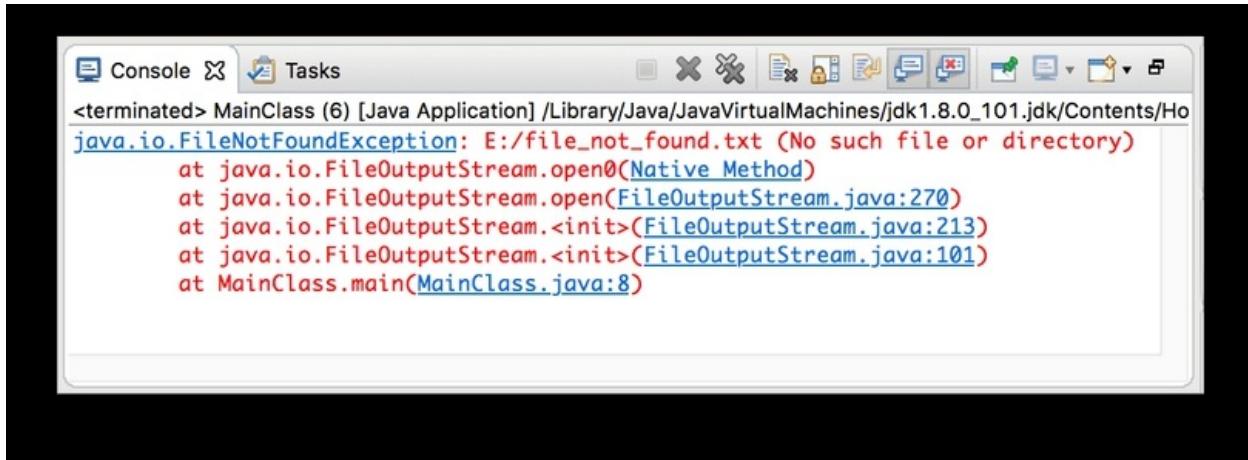
3. *printStackTrace ()*

Here is a method that helps you print the contents of the *toString ()* method up there , along with the *Stack Trace* log lines too.

Let's experiment together.

```
try(FileOutputStream outputStream = new  
FileOutputStream("E://file_not_found.txt")) { outputStream.write(65);  
} catch (FileNotFoundException e) {  
e.printStackTrace();  
} catch (IOException e) {  
e.printStackTrace();  
}
```

And the end.



The screenshot shows a Java IDE's console window. The title bar indicates it is a Java Application named MainClass (6). The console output displays a stack trace for a `java.io.FileNotFoundException`. The error message is: "E:/file_not_found.txt (No such file or directory)". The stack trace shows the following calls:

```
<terminated> MainClass (6) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Ho
java.io.FileNotFoundException: E:/file_not_found.txt (No such file or directory)
    at java.io.FileOutputStream.open0(Native Method)
    at java.io.FileOutputStream.open(FileOutputStream.java:270)
    at java.io.FileOutputStream.<init>(FileOutputStream.java:213)
    at java.io.FileOutputStream.<init>(FileOutputStream.java:101)
    at MainClass.main(MainClass.java:8)
```

You have just finished watching part 3 of the series about *Exception* . This's lesson helps us to go deeper on how to use *try catch* to *catch exceptions* and release resources. We have another part about this *exception* knowledge , you should watch it.

Lesson 37: Exception (Part 4)

I inform you that this is the last lesson about this "series" *Exception* . So looking back we have the following lessons about *Exception* as follows.

- *Get to know Exception* - This is the opening lesson about *Exception* . The lesson will help you get started with the concept of ERRORS in Java. From there, understand clearly what is *Exception* . In addition, the lesson also helps you distinguish the types of *Exceptions* in the Java system.

- *Getting (trapping) Exception* - After understanding what an *exception* is, you will know how to catch or trap *Exception* . Knowing this knowledge, you can "threading" the logic of the program through the *try catch* tool . This threadbreaking is to ensure that your application is always "alive". even if it is affected by ERRORS.

- *Learn more about try catch* - Once you know how to catch *Exception* through *try catch* , you will learn the further use of this tool through the two concepts *Try Catch with Finally* and *Try Catch with Resource* . And I also take advantage of some useful methods of the *Exception* class in this lesson, to help you better

understand how to use this tool, and can apply those methods for this's lesson. .

This, we come to the remaining knowledge of *Exception* . The lesson will guide you to reach a realm of fearlessness with the *exception* . You can even "juggle" them, push them to another place to process, and then you can even create your own *Exception* .

If you find it interesting, then please join the lesson.

I. Throw - Throw An Exception

All of us up to now all understand *Exception* . But we are still just "waiting" for *Exception* to come passively. That means we're only just *try* and waiting (even unwanted) *catch* the error.

But sometimes in practice, there are situations where we want to more quickly throw an *Exception* . I use the word "toss" because it is exactly the meaning of the word *throw* : throw, toss, toss. This released *exception* can be a *Checked* or an *Unchecked* Exception. And the release of this *exception* is on your will. The use of *throw* is also quite simple. Wherever inside a method, or block of statements, that you want to throw an *exception* out , use *throw* as with the following exercise.

- Practice Using Throw To Launch An Exception

We start with the design of a program that allows the user to enter the employee's age, and then print the age he just entered.

The requirements of the program seem extremely simple, but as you know, if not smart, the application can "crash" if the user intentionally entered an invalid age data.

For this exercise we try to tailor an age input *method* as follows.

```
private static int enterEmployeeAge() {  
    Scanner scanner = new Scanner(System.in);  
    System.out.print("Please enter employee's age: "); int age = scanner.nextInt();  
    return age;  
}
```

You can see, methods *enterEmployeeAge ()* has a *return type* is *int* (returns year-

old had recently entered). And this method is also declared with the keyword *static* , so we can call it from *main () method* (which is a *static* method). To understand why *enterEmployeeAge ()* has to declare *static* so you can review this *lesson* .

Next, in the *main () method* , to ensure the application doesn't crash, we should wrap the call to *enterEmployeeAge ()* inside a reasonable *try catch* .

```
public static void main(String[] args) {  
    try {  
        int age = enterEmployeeAge();  
        System.out.println("Age you entered: " + age);  
  
    } catch (InputMismatchException e) {  
        System.out.println("Age you entered isn't valid. Error: " + e.toString()); }  
}
```

From the previous lessons, you can easily guess that if you enter a 28 year old for example, the application will run fine. But if the user excuses to enter 28a , very quickly the application will report the error to the user immediately! Oh how much I love *try catch* .

Please enter employee's age: 8f

Age you entered isn't valid. Error: java.util. InputMismatchException

But if the user still has an excuse, enter an age with a negative value, -5 for example. Our application is still a bit "*stupid*" to leave this loophole.

Please enter employee's age: -5

Age you entered: -5

I do not have this age! So to upgrade the smartness of the application, we will take advantage of the *throw* keyword . As I said, this keyword will be utilized so that we can issue an *exception* whenever we want, in this case we should throw an *Exception* when the user enters a negative age. You can see your code added to the *enterEmployeeAge () method* as follows.

```
private static int enterEmployeeAge() {  
    Scanner scanner = new Scanner(System.in);  
    System.out.print("Please enter tuổi employee: ");  
    int age = scanner.nextInt();  
    if (age < 0) throw new InputMismatchException("Age entered can't bellow 0.");
```

```
return age;  
}
```

You can see the additional code highlighted above. I explain a little bit. Inside *enterEmployeeAge ()*, once checking that the age entered by the user is less than 0, the application will *throw* any *Exception*, in this example *InputMismatchException* is thrown. And you can understand that you can throw any *exceptions* you want. In addition to releasing an *Exception*, you can define a String type message and pass the *Exception*'s *constructor* as above code. Then in place call *enterEmployeeAge ()*, if that place has *try catch* It is reasonable, and meets the conditions that the *Exception* is released (entry age is less than 0), as in the previous lessons, the logic of the application will be broken on the *catch* block.

Now, with this upgrade, if the user enters a negative value, look.

Please enter employee's age: -5

Age you entered isn't valid. Error: java.util. InputMismatchException: Age entered can't bellow 0.

Through this practice, you have understood how to use *throw*, right.

Before going to the next section, I have a note that. With the above example, when we throw an *Exception*, we choose *InputMismatchException*. This is an *Unchecked Exception*. Therefore, the call to the method that has thrown this *Exception* is not forced by the compiler to add a *try catch*. But if you experiment with a *Checked Exception*, then there will be many things worth mentioning, and also related to the concept of *throws*, so I will spend this case and generally in the next section.

II. Throws - Throwing Exception For Elsewhere Handled

Pay attention, don't be mistaken. The above item mentioned about an *exception* voluntarily through the *throw* keyword. And this section is about *throws* (with an extra s).

throwing

talking

Other than *throw* that you can use inside a method or block of code. *Throws* are used as soon as you declare a method.

Throws is used when you don't want to have to build a *try catch* inside a method, you "push the responsibility" to *try* this for some outside method that calls it to *try* it for you.

Let's go to the following exercises to get the most out of *throws* , and also the good use of *throw* and *throws* together .

- Practice # 1: Using Throw To Launch a Checked Exception

Are you wondering why *throws* are talking about the title of a *throwback* practice ? You just try the following steps to see the real problem. I will get back the source code of the above exercise. In the method *enterEmployeeAge ()* , we try to replace *throwing* an *InputMismatchException* , with a certain *Checked Exception* . I will try with *IOException* . The code for this method will look like this.

```
private static int enterEmployeeAge() {  
    Scanner scanner = new Scanner(System.in);  
    System.out.print("Please enter employee age: ");  
    int age = scanner.nextInt();  
    if (age < 0) throw new IOException("Age entered can't bellow 0.");  
    return age;  
}
```

After releasing a *Checked Exception* as above, you easily realize that the system will ask you to do one of two things as follows.



The first option *Add throws declaration* will declare a *throws* for this *inputAgeEmployee ()* method. And as I said, if the method has declared *throws* , it does not need to *try to catch* anything anymore, but can push the responsibility of this *try catch* to some other method. The second option *Surround with try / catch* , you also know from the previous lessons, it will help wrap this code with *try catch* block . We should choose the first option in this situation, which both meets the requirements of the lesson and makes it easier to manage the code, because you just released an *exception* and tried to *catch it*. it just looks a bit disgusting.

With the first option, our code will look like this.

```
private static int enterEmployeeAge() throws IOException {  
  
    Scanner scanner = new Scanner(System.in);  
    System.out.print("Please enter employee age: ");  
    int age = scanner.nextInt();  
    if (age < 0) throw new IOException("Age entered can't be below 0."); return age;  
  
}
```

With the push of responsibility to *try to catch* this *Checked Exception*, the method calling *enterEmployeeAge()* will report a familiar error as follows.



I bet you already know what to do. And here, I have corrected it all you want, *try catch* with *IOException*.

```
public static void main(String[] args) {  
    try {  
        int age = enterEmployeeAge();  
        System.out.println("Age you entered: " + age);  
  
    } catch (IOException e) {  
        System.out.println("Age you entered isn't valid. Error: " + e.toString());  
    }  
}
```

Did you understand how to use *throws* to push the *try catch* responsibility away?

- Exercise # 2: Using Throws To Push the Exception Away

With the exercise above you probably already understand how to use *throws*. However, I want you to do one more example to focus on *throws* alone, with no *throw* appearing in this.

Let's go back to building the *try catch* situation like this exercise of the previous lesson. Only difference this time we build a separate file writing method and in this method there is no *try catch* at all.

Now, let's start with building the file recording method (the code for the file is the same as the exercise the day before).

```
private static void writeFile() {  
    FileOutputStream outputStream;  
    outputStream = new FileOutputStream("E://file.txt");
```

```
outputStream.write(65);
outputStream.close();

}
```

We will get an error again.



When you click on the bulb icon on the left, you already know that you should select this option.

And when fully *throws* are, this method should look like this.

```
private static void writeFile() throws FileNotFoundException, IOException {
    FileOutputStream outputStream;
    outputStream = new FileOutputStream("E://file.txt");
    outputStream.write(65);
    outputStream.close();

}
```

Finally, it is obvious where the call to this method will either force the full *try catch* , or *throws on the* next method.

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

    try {
        writeFile();
    } catch (FileNotFoundException e) { // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

III. Create Your Own Exception

At this point, we all have reached the highest peak of understanding and using *Exception* . When all of us then create an *exception* for ourselves.

Building our own *Exception* will be called with a short name: *Custom Exception* . *Custom Exception* is actually a certain *Exception* , it is given a more reasonable name in your application context (instead of the existing *Exception* names that

you have been familiar with). Then through the *Custom Exception*, you can adjust the error reporting, to bring more clarity to the application.

Before officially creating a *Custom Exception*, let's go through these important ideas first.

- All *Custom Exceptions* must be children of the *Throwable* class (you can review the knowledge of lesson 37 if you forget what class *Throwable* is).
- If you want to create a *Custom Exception* and want the system to error when not trying to *catch* it, then create a class and inherit from the *Exception* class (you can see in lesson 37 to know what class *Exception* is). And of course the *Exception* class is a child of *Throwable*, so if you do this, you will satisfy the other name.
- And if you want to create a *Custom Exception* And without the system error, you just create a class and inherit from the *RuntimeException* class (*RuntimeException* is also a child of *Throwable*, and the above link also talked about this class).

The above ideas are only children, right? Let's go to the practice together.

- Practicing Self Creating An Age Test Exception We will redo a program to enter employee age and check age as above. But instead of using *IOException* to check and throw an error, looking nothing, we should create a true *exception* in the case of this exercise. This *Custom Exception* will be of type *Checked Exception*. And this *Custom Exception* can be adjusted to make the error more visible.

The *Custom Exception* we are talking about will be named *AgeCheckingException*. Create a new layer as follows.

```
public class AgeCheckingException extends Exception {
```

```
    public AgeCheckingException(String message) { super(message); }
```

```
@Override
```

```
    public String getMessage() {  
        return "Error when entering age: " + super.getMessage(); }  
    }
```

Very simple. *This Custom Exception* will inherit from *Exception* class . So this is a *Checked Exception* already. You should also build a *constructor* for it to take in a message and then pass the message to the parent class through the *super ()* call . Finally, *Custom Exception* will *veto* the method *getMessage ()* to return more information.

We will build the *enterEmployeeAge () method* as follows. Maybe you have understood and are familiar with the following ways of using *Exception* , so I don't explain more.

```
private static int enterEmployeeAge() throws AgeCheckingException {  
    Scanner scanner = new Scanner(System.in);  
    System.out.print("Please enter employee age: ");  
    int age = 0;  
    try {  
  
        age = scanner.nextInt();  
        if (age < 0) throw new AgeCheckingException("Age entered can't bellow 0.");  
  
    } catch (InputMismatchException e) {  
        throw new AgeCheckingException("Age entered must be a number");  
    }  
    return age;  
}
```

And then in the *main () method* where we call *enterEmployeeAge ()* we will *try to catch* as follows.

```
public static void main(String[] args) {  
    try {  
        int age = enterEmployeeAge();  
        System.out.println("Age entered: " + age);  
  
    } catch (AgeCheckingException e) {  
        System.out.println(e.getMessage());  
    }  
}
```

The results will be as follows if you intentionally entered a negative number.
Please enter employee age: -5
Error when entering age: Age entered can't bellow 0. Or when entering a

number with characters.

Please enter employee age: 9s

Error when entering age: Age entered must be a number

So at the end of this's lesson, we have also finished with the interesting knowledge about *Exception* . The following lesson will be another interesting knowledge of Java, which is knowledge of *Thread* .

Lesson 38: Thread Part1 - Thread Concept

Last time there were many of you asking yourself when Java knowledge will end. Well, I also want to inform you that we are getting close to the final lessons. However, I also want you to know that, in fact, this ending is just in terms of my lessons. There is a lot of knowledge about Java, I hope to talk more about other aspects of Java in other topics. And while the lessons will not be able to keep up with the speed of your desire to learn, I hope you should also "ready" for you with other valuable Java learning materials.

Back to lesson about *Thread* . As I said from lesson 1, as soon as you know what Java is, I also mentioned one of the strengths of this language, which is *Support for multithreads (Multithread)* very good. And it is not until this's lesson that we come together to clarify this robustness of Java together.

I. Thread Concept, Or Multithread

Thread or *Multithread* both have the same meaning in this lesson's knowledge. *Thread* Vietnamese translation is *luồng* , and *Multithread* is *đa luồng* .

The thread here is the processing *thread* of the system. And because a good reason to let *Thread* birth is also to let the application can control multiple *Thread* various concurrently, which many *thread* and thus also means *Multi Thread* , or *Multithread* . That is why knowledge *Thread* or *Multithread* just one.

The role of *Thread* or *Multithread* is of course something related to *Multithreading* , *Multitasking* . In particular it will support system we split the task into several application *flows* (or *thread*), and the system will handle the *flow* of this *simultaneously* . So if according to what we are familiar with Java so far, it is if we have tasks A , B , C , with the old coding ways, the system will

always process the tasks sequentially. Now let's say A will be processed first, then B , And finally to C . But after this's lesson, if we organize such that each A , B, and C are each *thread* , it would be great because we can *absolutely get the system to handle both A, B and C at the same time* .

As you type each letter into the search box at the top of the screen, you expect to see location suggestions that will continuously update with each word you type below. As shown above. You see that, if there is no *Thread* , the system will wait for you to finish typing a word, then start searching and suggesting locations related to that word, and while the system is searching, you cannot type. get the next word, because with this way, at the same time the system only responds to one thing, either to receive the characters you type or search for, over.

But with the application of *Multithread* , we will have 2 *Threads* running in parallel, one *thread* receiving user input, The other *thread* are based on the entered data and search, this makes the user experience complete, the system is smooth, fast, and not jerky.

You can see yourself illustrating the response of the system to each search case as shown in the example above in the figure below. In it, to find results " *Harry Potter and a random episode*" appears in the list of suggestions. Then with the *No Multithread* column on the left, if you type a word and wait for the system to search, the results will take a long time compared to the *MultiThread* column on the right, your typing and the search system is separated, and real show up in a parallel way, is very quick and pleasant.



II. Distinguishing Related Concepts

This section I add, for newcomers to *Thread* (even with you already understand *Thread* already) to avoid freaking out when reading documents related to

knowledge this. Then you will be confused with the following concepts: *Thread*, *Multithread*, *Task*, *Multitask*, *Process*, *Multiprocess*.

First, at the most extensive level, you know that this's computer systems are all about multitasking capabilities. Multitasking is about handling multiple tasks at the same time. This multitasking is becoming more and more popular and interested as computers all have machines with multiple CPUs. Thus, the task that the system needs to handle is usually called the *Task*, and a multitasking system is called *Multitask*.

From the need to handle *Multitask*, the new system to ngia the *Process*. *Process* is understood as a program. When your application is launched, we will be creating a system of *Process* and applications the *Process* that. The system allocated resources independently system together. And by the same time can have multiple *Process* (or multiple applications) run parallel to each other, what we call the *Multi-Process* is *Multiprocess*.

that will be implemented and managed within will manage more than one *Process*, each *Process* so

When a *Process* was created, the application within the *Process* which could create many *Thread* else. Basically, *threads* will be managed by the system like *Process*, ie they can be run in parallel, so there is a new concept of *Multithread*. But the *thread* inside a *Process* can only operate within the limits of the *Process* there. The *thread* will be using resources like the *Process* of it is allowed. But one difference is *Threads* very light and can be easily shared with other shared resources within a *Process*.

Just like that. And despite all the rambling of such concepts, but remember, in this lesson we are only interested in one kind of concept, which is *Thread*.

III. When to Use Thread

This question is easy to answer, but if you think about it carefully it contains more information than we think.

First of all, I mentioned a long way above that, if you want to have multiple tasks that want to work together, then use *Threads*. For example, you just want the application to get user information into the dialog box, just call the server to check the input results. Or if you program a game, you just want the game to

receive control signals from the keyboard, along with drawing characters on the screen according to that control, along with controlling obstacles, or controlling projectiles, score displays, etc.

The second use of *the Thread* is also akin to want to handle these tasks simultaneously, but when you wish to control the *flow* in a certain consistency. For example, you start multiple *Threads* in an application, but you want other *Threads* to be started but wait, but wait for a certain *thread* to finish before being active. In Volume 3 of the series on *Thread*, I will clarify this usage of *Thread*.

The third point in using *Thread*, also does not want to handle concurrent tasks, but when this is the situation when you want the application to execute too large tasks. Big here is usually big in terms of time. For example, when an application has to download a file from a server, or have to do something that its completion time is not instantaneous. Then without a *thread*, these lengthy tasks can seriously affect the user experience.

We will come to the example of *Thread* through the following exercise to help you better understand.

- Building Random Number App

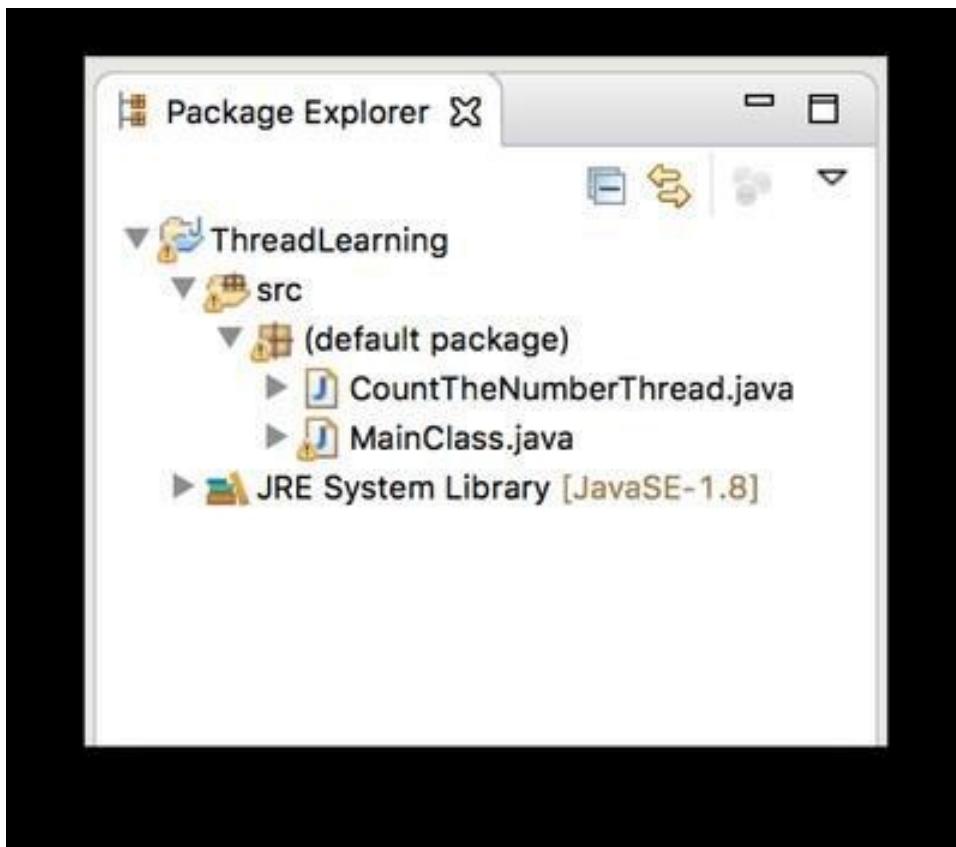
This exercise gives you a good idea of what a *Thread* is and how it works. I do not want you to fully understand the code of this exercise, but I hope you will try the code, and successfully execute the program. If there is any error occurring to you, please contact me or leave a comment below this's lesson.

Imagine that this exercise is building a game. In this game there is a turntable, on which there are numbers from 0 to 100. The player will start rotating the turntable, after the turntable is turned, the player does not see the numbers on it, until they stop the turntable, which number is right at the player is the number that is play select.

Well, our app won't build on that kind of a complete game. Instead we will focus on the logic of the game. We will replace the turntable with a command to type any character on the *console*. After receiving the character just typed, instead of having a spinner, we will launch a *Thread*, this *Thread* will in turn "spin" the numbers, so that it counts from 0 to 100 and then again. dial back to 0. Our *thread* runs until the player enters a character from the console and the *thread* ends. The number that *Thread* just "stopped" it is the number chosen by the player.

You have understood the requirements of building this game, right. To get started in game building, you should *create a new project yourself*. You can name this project *ThreadLearning* also. Then *create a new class containing the main () method* in it.

Next, create a new class by yourself called *CountTheNumberThread*. Wait, please code anything for this class. This class is the *Thread* responsible for "spinning" the numbers as we discussed. In general, our application has the following structure.



Now, code the *CountTheNumberThread* class body as follows.

```
public class CountTheNumberThread extends Thread {  
    private int count = 0;  
    private boolean isStop = false;  
    @Override  
    public void run() {  
        while(!isStop) {  
            count++;  
            if (count > 100) {  
                count = 0;  
            }  
        }  
    }  
}
```

```

}
}
}

public void end() {
    isStop = true;
}

public int getCount() { return count;
}
}

```

Although you may not understand *Thread well* , but I also want to explain a little, so to the next lesson, you will easily approach *Thread* .

First, the class *CountTheNumberThread* wants to be a *Thread* , it must inherit from the *Thread* class (and there is another way to turn a class into *Thread* that we will talk about in the next lesson). Then inside this class must *override* the *run () method* . The main method of code inside *run ()* will be a *Flow* , *Flow* will be executed in parallel systems with the *Flow* executing another if present in the system. That's all, apart from the two big ideas I mentioned like this, the remaining declarations of *CountTheNumberThread* are all too familiar, so I did not present much.

At this point, *CountTheNumberThread* is just a class. Want to launch this class to become a *Thread* , I invite you to the code in the *main () method* as follows.

```

public static void main (String [] args) {
    Scanner scanner = new Scanner (System.in);

    // Wait for the user to press a key to start System.out.println ("Press any key to
    start"); scanner.nextLine ();

    // Declare & Launch CountTheNumberThread as a passed Thread // start ()
    method
    CountTheNumberThread countingThread = new CountTheNumberThread ();
    countingThread.start ();

    // Wait for the user to press a key to finish System.out.println ("Press any key to
    stop"); scanner.nextLine ();

    // Stop Thread and see which number is currently "spinning" countingThread.end
}
```

```
());  
System.out.println ("Lucky Number:" + countingThread.getCount());  
}  
}
```

Looking at the above code, you can see a little difference between using a *Thread* versus another normal class. What if instantiating a normal class you know versus a *thread* is the same. Then with a *thread* , to launch the *thread* there (ie you turn it into a *stream* in the system), you must call the method *start ()* of it, this method is built in class *Thread* . When *start ()* is called, the parent *Thread* will launch the code inside the *run () method* you declared inside the *CountTheNumberThread* . . And so receiving input from the keyboard in the *main () method* will parallel the loop inside this *run ()* method, no code has to wait for any code.

If you execute the program, enter a character in the console to start "dial in" and enter again to finish, you will get a lucky number of your own as follows.

Press any key to start

f

Press any key to stop

r

Lucky Number: 69

Opps! I dialed 69 , what about you. Good luck to you. • Exercise

For this exercise, I want you to try to ignore the *Thread* inheritance of the *CountTheNumberThread* class , then the *run () method* of this class will no longer have the *override* keyword , so you have turned this class into a normal class. and *run ()* is just another normal method. Then at *main ()* don't call *countingThread.start ()* anymore, but call *countingThread.run ()* . Ie you don't use *Thread* in this case. You can re-execute the application and compare whether applying *Thread* to this game will be different from not applying *Thread* how will it be

We have just come across a fairly new and interesting understanding of Java about Multitasking, namely *Thread* . This's lesson is just the initial familiarity, *Thread* has many other interesting knowledge that I will in turn present in the upcoming lessons.

Lesson 39: Thread (Part 2)

After the first episode about Thread was released, I received a lot of sharing and feedback from you. I feel your great interest in this knowledge. This is really interesting. Actually, I used to be very interested in approaching Thread.

Although only a small knowledge in the Java knowledge sea, but Thread is bringing a new wind, a new ability for us to build multitasking, more powerful, practical applications, make the most of them. more system performance. And more specifically, after knowing what Thread is, we can start to learn about building a Java game already.

So this, we will continue to reinforce our interest in Thread by going more specific about it, we will talk about ways to declare and instantiate a Thread.

I. Create A Thread

In *the previous lesson*, you were familiar with a way to create a thread already. But I want this's lesson, you ... forget the previous lesson's knowledge, let's go over and over for it.

In Java, *there are two ways for you to create a Thread* . Both methods have the same frequency of use (according to my observations). It is your job to know both ways. Why must know both ways? In addition to knowing all to be able to declare and use, you also have to know to also read and understand the source code of others when they are not using the same as you.

So what are the two ways to create Thread.

1. Method 1 - Inheriting From Thread Class

This way from the previous lesson... oh I told you to forget it. So, this way you do the following.

- You create a new class and inherit this class from *Thread* parent class . In that newly created class, you *override* the *run () method* .

Finally, elsewhere, when you want to create a Thread from this class, you declare an object for it, and then call its *start () method* to start Thread initialization.

It's simple, right. Didn't expect Thread knowledge to be so easy. As you know briefly from the lesson yesterday that to declare a class is thread, then simply inherit from the parent class *Thread* , the method *run ()* inside the class will

become a *stream* handled by the system when somewhere outside calls the *start () method* of this class (actually *start ()* is the inheritance from the superclass *Thread*).

Together we come to practice to understand better.

2. Practice Creating A Thread By Inheriting Thread Class

In this exercise we test create a *10 second countdown Thread* . When start, Thread will start to print console value *10* , every second passed Thread will reduce this number by one unit and print to console again, until decreasing to value *0* Thread will print "*Time Out*" .

You can reuse the project you created in the previous lesson, this you create a new class called *CountDownThread* . This class will inherit from the *Thread* class as I said in the bullet points above as follows.

```
public class CountDownThread extends Thread {  
    @Override  
    public void run() { // We will code later }  
}
```

A framework for Thread just like that. As mentioned, *CountDownThread* when started will start counting down from *10* seconds, up to *0* seconds will display the string "*Time Out*" . The number of seconds to the console, you know, I just revealed that to make this number only updated and displayed every second, we use the *Thread.sleep (1000) method* . This method causes the running threads to "*sleep*" for a period of milliseconds, in which case we pass in *1000* milliseconds, which is *1* second. After sleeping for the maximum amount of time allowed, Thread will "*wake*" action. Note that you must *try to catch* this *Thread.sleep* and continue with its () method with a *Checked Exception* named *InterruptedException* . And I will talk about the *Thread.Sleep () method* in the following lesson. And here is the complete code of *CountDownThread* .

```
public class CountDownThread extends Thread {  
  
    @Override  
    public void run() {  
        int count = 10;  
        for (int i = count; i > 0; i--) { System.out.println(i);  
    }
```

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}  
}  
}  
System.out.println("Time out");  
}  
}
```

To start the newly created Thread, we will call its *start () method* as follows.

```
public static void main(String[] args) {  
    CountDownThread countDownThread = new CountDownThread();  
    countDownThread.start();  
}
```

¹⁰ And here is the console screen of the "timer" we just
9 created. Every second a number will appear until the
8 word "Time Out" appears at the end of the program (and the
7 end of *CountDownThread*).

6

⁵ 3. Method 2 - Implement From Interface Runnable

⁴ If the above method, you must inherit from *Thread* class , then₃ this way you
implement an *interface* named *Runnable* . This

2 way you do the following.

1

Time out - You create a new class and implement this class with *Runnable* . In
that newly created class, you override the *run () method* . Finally, somewhere
else, when you want to create a Thread from this class, you first declare an
object for it, then you declare another *Thread* object and pass the object of this
class to the constructor. by *Thread* . When the method *start ()* of class *Thread*
just created is called, the method *run ()* within the class of derivatives *Runnable*
will be called to form a *flow* in the system.

It sounds more complicated than the first method up there, right. But you should also give it a try by going to the following exercise.

4. Practice Creating A Thread By Implementing From Interface Runnable

We will still rebuild the example of a Thread countdown 10 seconds up there by this 2nd.

With this way, you just need to edit a little bit in your *CountDownThread* class , so that from *extends Thread* to *Runnable implements* is done. You see the following code will clear. public class CountDownThread implements Runnable {

```
@Override  
public void run() {  
    int count = 10;  
    for (int i = count; i > 0; i--) {  
        System.out.println(i);  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) { // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }  
    System.out.println("Time out");  
}
```

The problem of declaring a Thread is not much different between the two, right. The more difference will be in how Thread is started. The code in the *main () method* will have to be changed as follows.

```
public static void main(String[] args) {  
    CountDownThread countDownThread = new CountDownThread();  
    Thread thread = new Thread(countDownThread);  
    thread.start();  
}
```

Please execute the program again. The result of these two methods of doing both gives the same result.

II. Applying Anonymous Class Knowledge In Creating New A

Thread

If you have forgotten what class Anonymous is, then you can read the lesson in [*this link again*](#).

And if you wonder what Thread has to do with Anonymous class? Then I will explain briefly like this. Threads are basically seen as a lightweight way for the system (and us) to do parallel tasks. And to make that lightness even more compact (in terms of code management), the combination of Thread and Anonymous class would be a good solution for this idea. Because then, we will not need to explicitly declare a Thread class at all, simply construct an Anonymous class, and start it.

It is quite common to combine Thread and Anonymous class, and they have combined these 2 names into a common name, called *Anonymous Threads*. Let's see the following ways to "*anonymize*" a Thread. I use the example Thread countdown above for you to see.

1. Creating An Anonymous Thread From Inheriting Thread Class

Let's recreate a Thread from extending the Thread class, but "*anonymize*" it as follows.

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

```
    Thread countDownThread = new Thread() { @Override
        public void run() {

            int count = 10;
            for (int i = count; i > 0; i--) {
                System.out.println(i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) { // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            System.out.println("Time out");
        }
    };
    countDownThread.start();
}
```

See, above is an Anonymous Thread. At this point, many of you will wonder if Anonymous Thread actually helps to make the code management more compact or not. Then I have a few more ideas to discuss as follows.

In fact, whether using Anonymous Thread compared to a normal Thread makes the code compact or not, depending on how each person's code looks. You see, with the practice of building a normal Thread over there (I will call Thread-normal-Thread for short), you must build a very *obvious* *CountDownThread.java* class , this code has something very explicit. . As for the example code in this section, we have created a *countDownThread* object not from the *CountDownThread* class or from the *Thread* class , but from *an Anonymous class inheriting from the Thread class*. Please. This declaration helps to reduce the need to create another Java file, we simply declare and use it, in addition, Anonymous Thread can also use the members of the class that contains it.

Another difference between declaring a Thread and an Anonymous Thread is that, with Thread you can build a constructor for it, so you can pass in Thread certain variables that serve application logic. And Anonymous Thread has no constructor, so you may have to use the global variable of the declared class.

But you should also consider, although using Anonymous Threads is quite quick and convenient, they can cause code in this usage class to swell, harder to manage if there are so many Anonymous Threads. here it is.

Back to the knowledge of Anonymous Thread, with the above code, we can write even more concise. By not having to declare the object name, *start () can always be done*, like this.

```
public static void main(String[] args) {  
    new Thread() {  
        @Override  
  
        public void run() {  
            int count = 10;  
            for (int i = count; i > 0; i--) {  
  
                System.out.println(i);  
                try {  
  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                }  
            }  
        }  
    }.start();  
}
```

```

// TODO Auto-generated catch block
e.printStackTrace();
}
}
System.out.println("Time out");
}
}.start();
}

```

2. Create An Anonymous Thread By Implementing From Interface Runnable

If you understand Anonymous Thread from the way of extending the Thread class above, then creating an Anonymous Thread from the *Runnable* interface can also be written on your own.

```

public static void main(String[] args) {

Runnable countDownThread = new Runnable() { @Override
public void run() {

int count = 10;
for (int i = count; i > 0; i--) {
System.out.println(i);
try {
Thread.sleep(1000);
} catch (InterruptedException e) { // TODO Auto-generated catch block
e.printStackTrace();
}
}
System.out.println("Time out");
}
};

Thread thread = new Thread(countDownThread); thread.start();
}

```

This code can also be more concise by removing the object declaration from the *Thread* class as follows.

```

public static void main(String[] args) {

Runnable countDownThread = new Runnable() { @Override

```

```
public void run() {  
  
    int count = 10;  
    for (int i = count; i > 0; i--) {  
        System.out.println(i);  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }  
    System.out.println("Time out");  
}  
};  
new Thread(countDownThread).start(); }
```

Or can be even more concise when there is no need to declare the Anonymous Thread object. But then you must pass this Anonymous class to the *Thread* as an argument. As follows.

```
public static void main(String[] args) {  
    new Thread(new Runnable() {  
        @Override  
        public void run() {  
  
            int count = 10;  
            for (int i = count; i > 0; i--) {  
                System.out.println(i);  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) { // TODO Auto-generated catch block  
                    e.printStackTrace();  
                }  
            }  
            System.out.println("Time out");  
        }  
    }).start();  
}
```

Hope knowledge of Anonymous Thread does not make you too headache. I invite you to the following exercise to be able to "*get used to*" more in creating a Thread.

- Exercise 1: Creating A 2 Thread Game Together Guess Numbers Exercise requirements are as follows. Create a game where the user can enter an integer between 1 and 100 . Then you build a Thread guessing the number, this Thread will direct random numbers in the range 1 to 100 . Every time a random number is given, the Thread will print out the console for the player to see. Thread will stop when randomized a number that matches the number that the player just entered, and printed out the number of "*guesses*" to get that number.

Note that, there are 2 Threads guess the same number, to "*contest*" to see which Thread "*guess*" the number of the fastest player.

Thread 1 has counted 49

Thread 2 has counted 15

Thread 1 has counted 34

Thread 2 has counted 15 after 68 times

Thread 1 has counted 85

Thread 1 has counted 71

Thread 1 has counted 3

....

Thread 1 has counted 35

Thread 1 has counted 89

Thread 1 has counted 15

Thread 1 has counted 15 after 360 times

To make it easier to imagine, I give the expected console result of the game like this. This result is based on the hard "guess" of the 2 Threads, to be able to know the player entered number 15 . And as shown below, *Thread 2* won with 68

guesses. *Thread 1* is "less intelligent" and continues to guess until the 360 guess.

I have two suggestions for you to focus only on the code for Thread, not having to worry about learning other code on the internet.

- To randomize a number from 1 to 100 , you code:
randomNumber = (int) (Math.random() * 100 + 1);

- The parent *thread* has a *setName () method* for you to name the running Thread, so you can use it to name the "*Thread 1*" , "*Thread 2*" . You can then print to the console by calling the name you *specified* with the *getName () method* .

Done, invite you to code. After the code is done, you can compare it with your answer. First, Thread guesses the number, I named it *GuessANumberThread* .

```
public class GuessANumberThread extends Thread {  
    private int guessNumber = 0; private int count = 0;  
  
    public GuessANumberThread(int guessNumber) { this.guessNumber =  
        guessNumber;  
    }  
  
    @Override  
    public void run() {  
        int randomNumber = 0;  
        do {  
            randomNumber = (int) (Math.random() * 100 + 1);  
            count++;  
            System.out.println(getName() + " has counted" + randomNumber);  
  
        try {  
  
            Thread.sleep(500);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        } while (randomNumber != guessNumber);  
  
        System.out.println(getName() + " has counted" + guessNumber + " after " +
```

```
count + " times"); }  
}
```

And this is the place to ask the user to enter a number to guess, and create 2 Threads to test. This place is the *main () method* .

```
public static void main(String[] args) {  
Scanner scanner = new Scanner(System.in);  
System.out.println("Enter a number for them to guess");  
int number = scanner.nextInt();  
GuessANumberThread thread1 = new GuessANumberThread(number);  
GuessANumberThread thread2 = new GuessANumberThread(number);  
  
thread1.setName("Thread 1"); thread2.setName("Thread 2");  
thread1.start(); thread2.start(); }
```

The result of program execution will be as shown above. • Exercise 2: Repeat Exercise 1 With Runnable

Please try to re-code the above guessing game using Thread implement from *Runnable* .

There is only one note that makes it easy to code that, to be able to call the name of Thread implement from *Runnable* , you cannot just call *getName ()* like *Exercise 1* , you must call *Thread.currentThread (). GetName ()* .

- Exercise 3: Reworking Exercise 1 With Anonymous Thread

This time you can re-code this number guessing game with anonymous Thread. This is how I do it, how are you doing.

```
public static void main(String[] args) {  
Scanner scanner = new Scanner(System.in);  
System.out.println("Enter a number for them to guess "); int guessNumber =  
scanner.nextInt();
```

```
Runnable anonymousGuessANumber = new Runnable() {
```

```
    @Override
```

```
    public void run() {  
        int randomNumber = 0;  
        int count = 0;  
        do {  
            randomNumber = (int) (Math.random() * 100 + 1);
```

```

count++;
System.out.println(Thread.currentThread().getName() + " has counted" +
randomNumber); try {
Thread.sleep(500);
} catch (InterruptedException e) {
e.printStackTrace();
}
} while (randomNumber != guessNumber);

System.out.println(Thread.currentThread().getName()
+ " has counted" + guessNumber + " after " + count + " times"); }
};

Thread thread1 = new Thread(anonymousGuessANumber);
thread1.setName("Thread 1");
Thread thread2 = new Thread(anonymousGuessANumber);
thread2.setName("Thread 2");
thread1.start(); thread2.start(); }

Above are all the ways you can create a thread. You see, Thread is easy to use or
not. Leave a comment below the lesson if you have any questions.

```

Lesson 40: Thread (Part 3) Life Circle

Step into this part of Thread, we will learn more about Thread, to see when you create a certain Thread, its life cycle will be like? What states does the thread go through during that lifecycle? Based on these states, how can threads synchronize, or can be understood as manually adjusting the priority of task execution between threads in the same process? We invite you to join us in this interesting knowledge this.

First of all, let's answer the first question.

I. What Is The Life Cycle Of An Object?

I just repeat a little bit, that's why we consider the "*life cycle*" of an object, when that object has a certain lifetime, and during the life of that object they I want to know how it is likely to go through many different states. Are those states of Birth, Aging, Sickness, or Death? ^^

In short, it is not everything we all talk about its life cycle, only objects that have a long enough life, and go through many states in the course of life, we will see. and Thread in this Java knowledge for example.

So what is the life cycle, or states within a life cycle for.

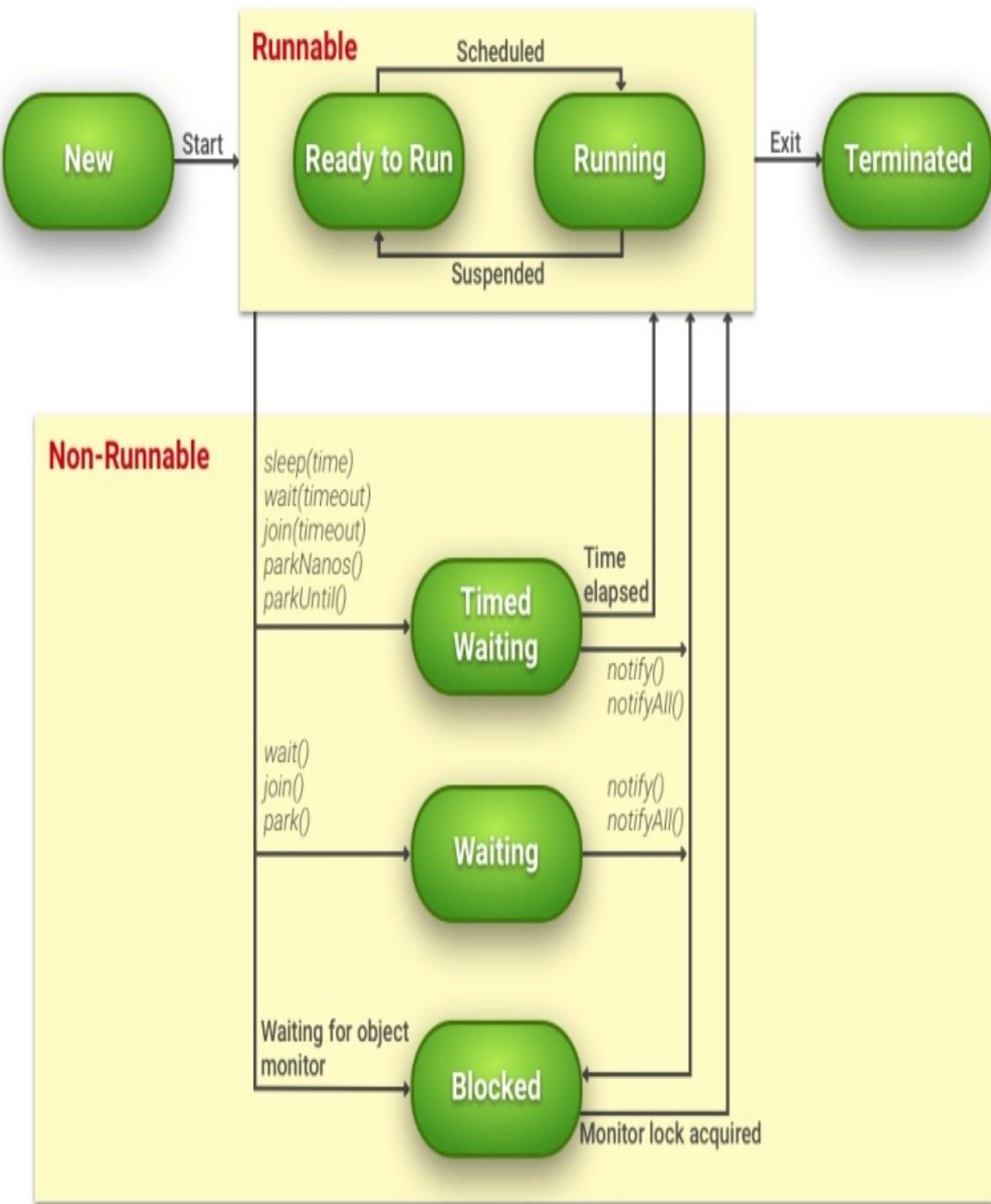
II. Why Should A Person Learn The Life Cycle?

One of the best reasons we should know the life cycle of an object, other than when it was born, and when it died. It is also quite important to understand the states that the object goes through during that life. When you understand the states of a life cycle, you will understand the object better, so that you can easily intervene in it, inserting those states with the most relevant tasks. The ultimate goal is to make our application more powerful, and even, smarter. For more details, please see more.

III. Understanding The Life Cycle Of Thread

Let's go back to the main part of the lesson, and together explore the Thread's lifecycle. First of all invite you to view this life cycle diagram through the following diagram.

1. Thread Lifecycle Illustration Diagram



This diagram builds upon the states defined in the `Thread` class's *enum* declaration. What is *Enum*, we will talk about in the next lesson. Basically, you can understand that *enum* helps us define a set of *constants*, and in this case *these constants are also states of Thread lifecycle*.

You can go into the *Thread* class to see what it's like to declare values inside an *enum*. You can go inside the *Thread* from Eclipse to see, or can see the following image.

```
public enum State {  
    * Thread state for a thread which has not yet started. []  
    NEW,  
  
    * Thread state for a runnable thread. A thread in the runnable []  
    RUNNABLE,  
  
    * Thread state for a thread blocked waiting for a monitor lock. []  
    BLOCKED,  
  
    * Thread state for a waiting thread. []  
    WAITING,  
  
    * Thread state for a waiting thread with a specified waiting time. []  
    TIMED_WAITING,  
  
    * Thread state for a terminated thread. []  
    TERMINATED;  
}
```

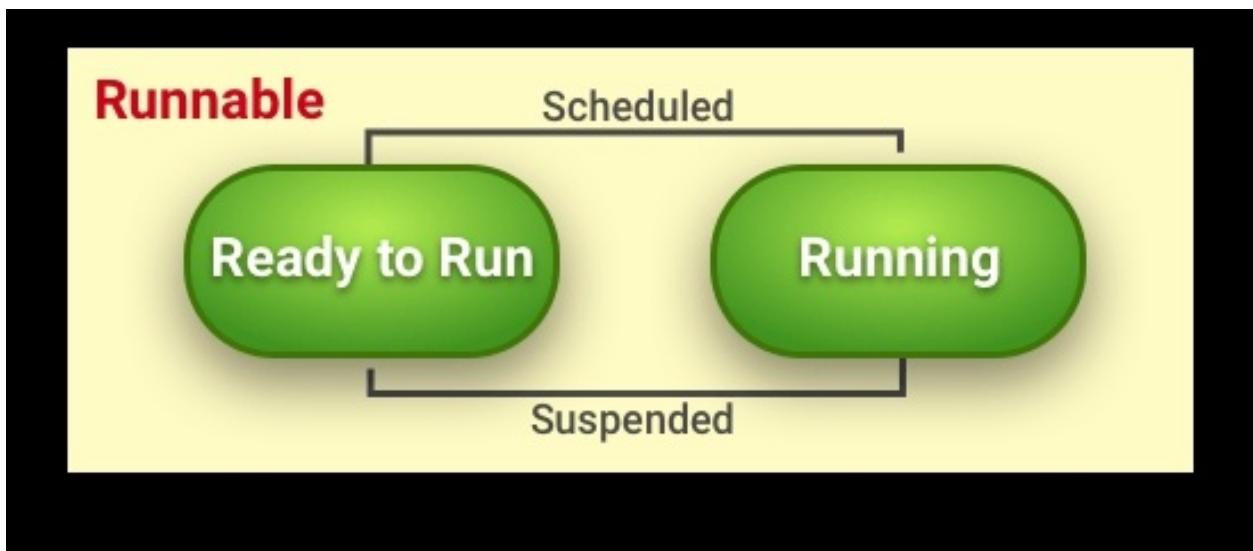
2. Life Cycle Description Of Thread

Like I said, the diagram or the *enums* inside a *Thread* have the most obvious state of this thread's life cycle. They are described in general as follows.

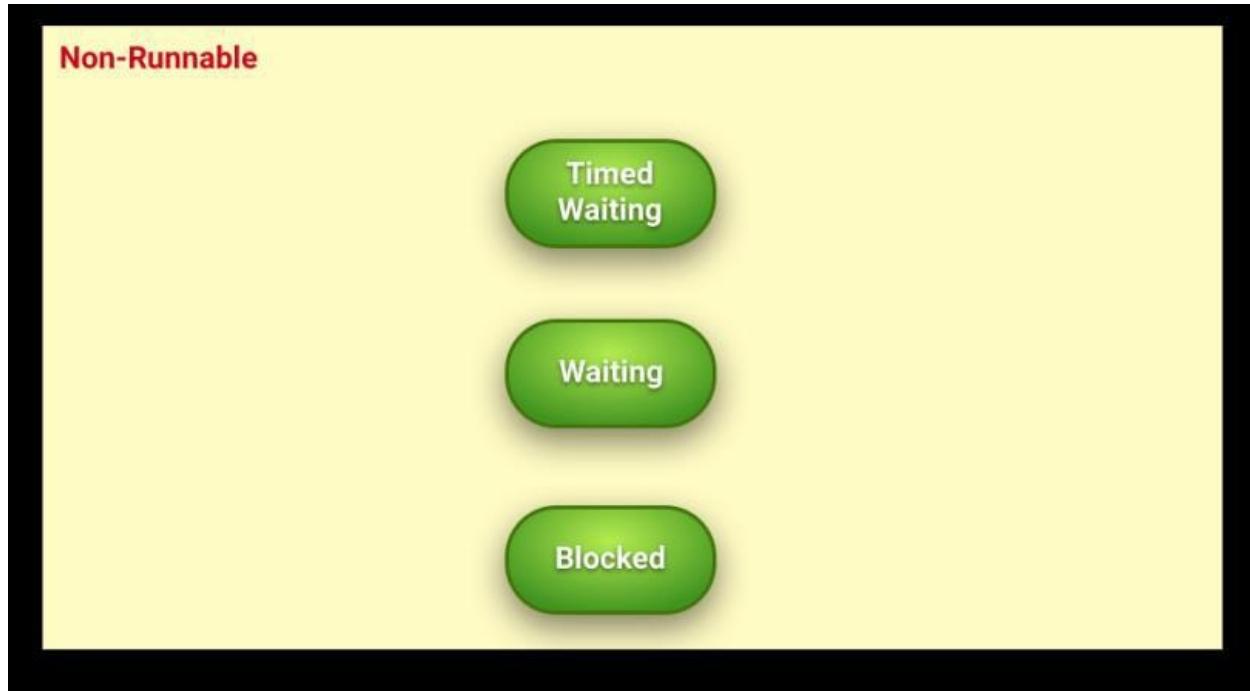
As soon as you create a new *Thread*, but have not yet called the *start () method*, its state will be *NEW*.



When you have called `start()`, that Thread will enter the *RUNNABLE* state, which puts the Thread in the queue to wait for the system to allocate resources and then launch it.



During the running thread, if there is any action, except the end of the thread's lifecycle, it will enter the state of *BLOCKED*, or *WAITING*, or *TIMED_WAITING*.



Finally, when a thread ends, it reaches the *TERMINATED* state .



The overview is so, and the details of each status, I invite you to the next section will clear.

IV. Statuses Within A Life Cycle

In this section we will take a closer look at each state. The main thing about this section is to help us understand when a Thread falls into a certain state. Thereby you can make use of it for specific purposes in your specific projects later.

1. NEW

This state is easy to understand, when you initialize a Thread, but have not called

its *start () method* , the thread will fall into the *NEW* state . Don't believe it, invite you to the following exercise.

- Practice # 1

In this first exercise, we will see if the state when a thread is initialized but the *start () method* has not been called is *NEW* .

In order to see the status of a Thread, we will call the *getState () method* . This method is built in the *Thread* parent class .

Please create a new Thread. It is fine to either *extend the Thread class* or *implement it from the Runnable interface* . Please name this Thread *MyThread* . This is how I build *MyThread* .

```
public class MyThread extends Thread {
```

```
    @Override
```

```
    public void run() {  
        System.out.println("Thread Start");  
    }
```

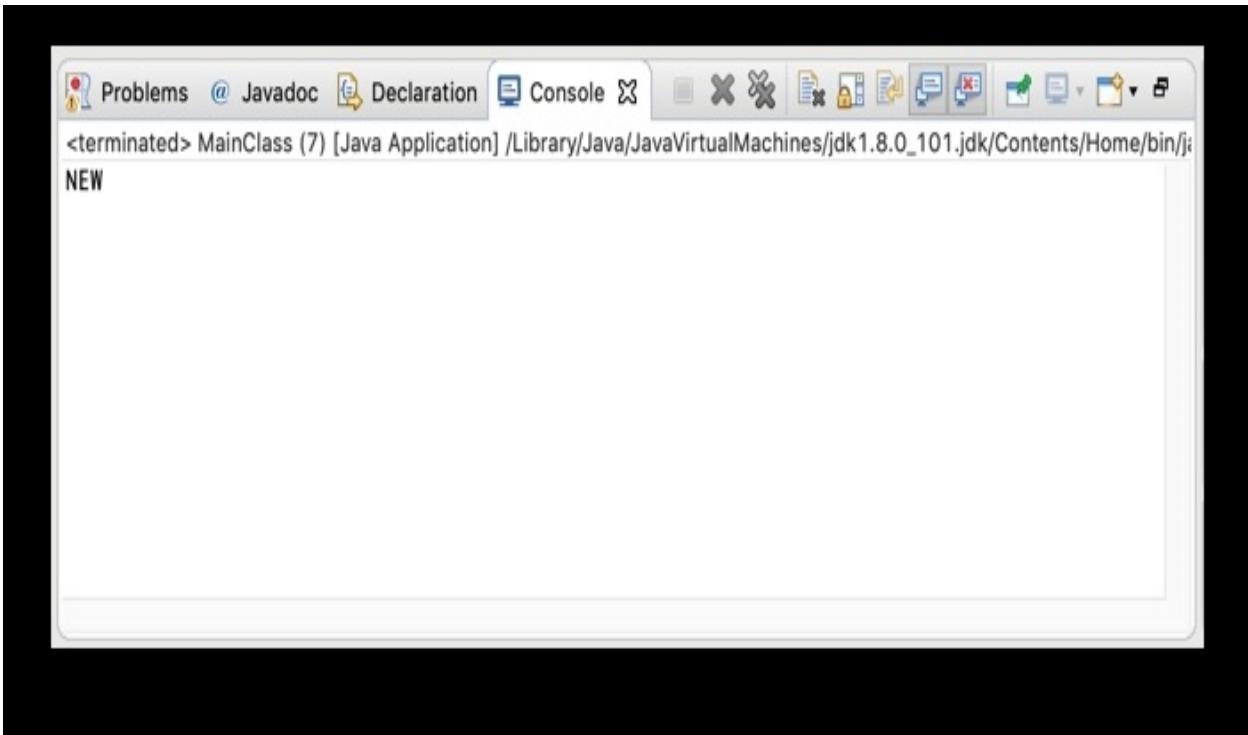
To see this *NEW* state , in the *main () method* , declare *MyThread* and then print *getState*

) without having to *start ()* it.

```
public static void main(String[] args) { MyThread myThread = new  
    MyThread(); System.out.println(myThread.getState());
```

```
}
```

And this is the "*finished product*" .



2. RUNNABLE

This state occurs when the thread has been called the *start () method*. Oh, you should also know a bit that is not *start ()* finish is the thread running right where it was waiting system resource allocation finished before starting a run. That is why inside this state seems to be divided into two sub-states, that is, *Ready to Run* - Waiting for resource allocation, and *Running* - Has officially run.

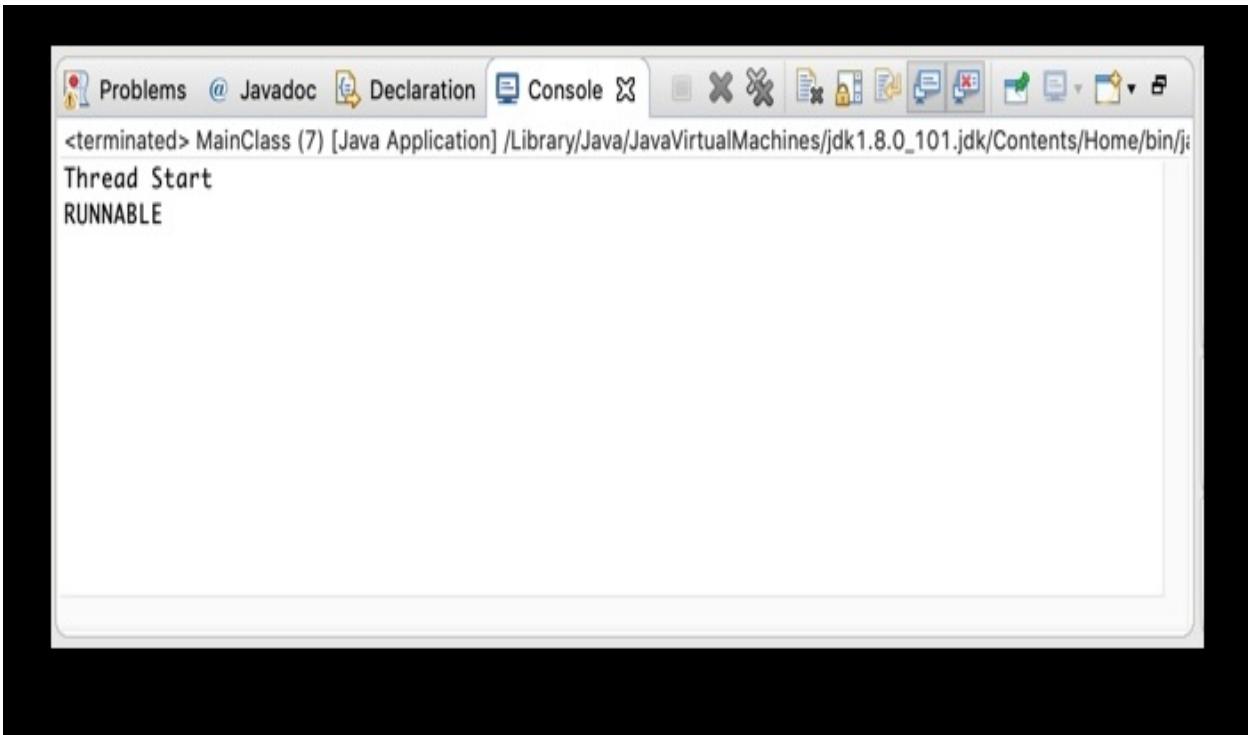
- Practice # 2

In this lesson we will try to call the *start () method* of *MyThread* in *Exercise 1* above. Then also call the *getState () method* right after that to see what the status of *MyThread* is now.

The code in the *main () method* is as follows.

```
public static void main(String[] args) {  
    MyThread myThread = new MyThread();  
    myThread.start();  
    System.out.println(myThread.getState());  
}
```

The results are printed to the console.



Note that the above code does not always print the *RUNNABLE* state to the console. Why? You can see that inside *MyThread* only print out the console string "*Thread Start*" , after printing this string, Thread will end its life cycle. Therefore there is a case where *geState ()* in the *main () method* will call when *MyThread* has finished, so *RUNNABLE* may not be printed (but a certain other state in the following items you will see) is so.

3. BLOCKED

A thread when it falls into *BLOCKED* state is when it is not eligible to run. How is not eligible to run? You can understand,? By the nature of the threads in an application are capable of running in parallel when they are *started ()* . Thus, it will happen that at a certain time, there will be more than one Thread having a "*scheme*" to edit a File or an object, we call the Files or objects are "*dispute*" These are shared resources. If this dispute occurs, it will cause the application to crash, possibly resulting in data loss or miscalculation. Therefore, there is a mechanism in Java that controls threads, which ensures that only one Thread can at any time interfere with a shared resource. This mechanism is related to the concept of *Synchronization*, which we will have a separate lesson about. So if this synchronization occurs, then only one Thread will have priority to use this shared resource, the remaining Threads? locked and have to wait for the other

priority Thread to finish using the resources before being run, these locked *Threads* will fall into *BLOCKED* state .

So in order to be able to see this state, we will get acquainted a bit with knowledge about *Synchronization* in the next exercise, and then in the next lesson we will talk more about it together.

Exercise Number 3

To practice this section, let's create a shared resource, which is a certain class, we name this shared class *DemoSynchronized* . In this class, there is a *static* method marked *synchronized* . This method is named *commonResource ()* . Do not focus on the *synchronized* keyword too, the next lesson will explain clearly. You just need to understand that this *commonResource ()* method marked *synchronized* will be "sponsored" by the system so that only one Thread can access it.

```
Let's just build before this class. public class DemoSynchronized {  
  
    public static synchronized void commonResource() { for (int i = 0; i < 100000;  
        i++) {  
        // Không làm gì cả, chỉ chạy vòng lặp để đảm // bảo phương thức này sống lâu  
        // một tí, // để cho có Thread dùng đến và các Thread // khác phải chờ đợi  
  
    }  
    }  
}
```

We then let *MyThread* (coded in the exercises above) have a chance to call *commonResource ()* . As follows.

```
public class MyThread extends Thread {  
    @Override  
    public void run() {  
        DemoSynchronized.commonResource();  
    }  
}
```

And then in the *main () method* , we will create more than one object of *MyThread* , namely 2 objects, you can also create 3 , or 4 *MyThread* for verification. After creating *MyThreads* , we both *start ()* them together, so that they call *commonResource ()* at runtime. Then you just need to "cavalier" call *getState ()* thereof.

```
public class MainClass {
```

```

public static void main(String[] args) { // Khai báo nhiều đối tượng của
MyThread MyThread myThread1 = new MyThread(); MyThread myThread2 =
new MyThread();

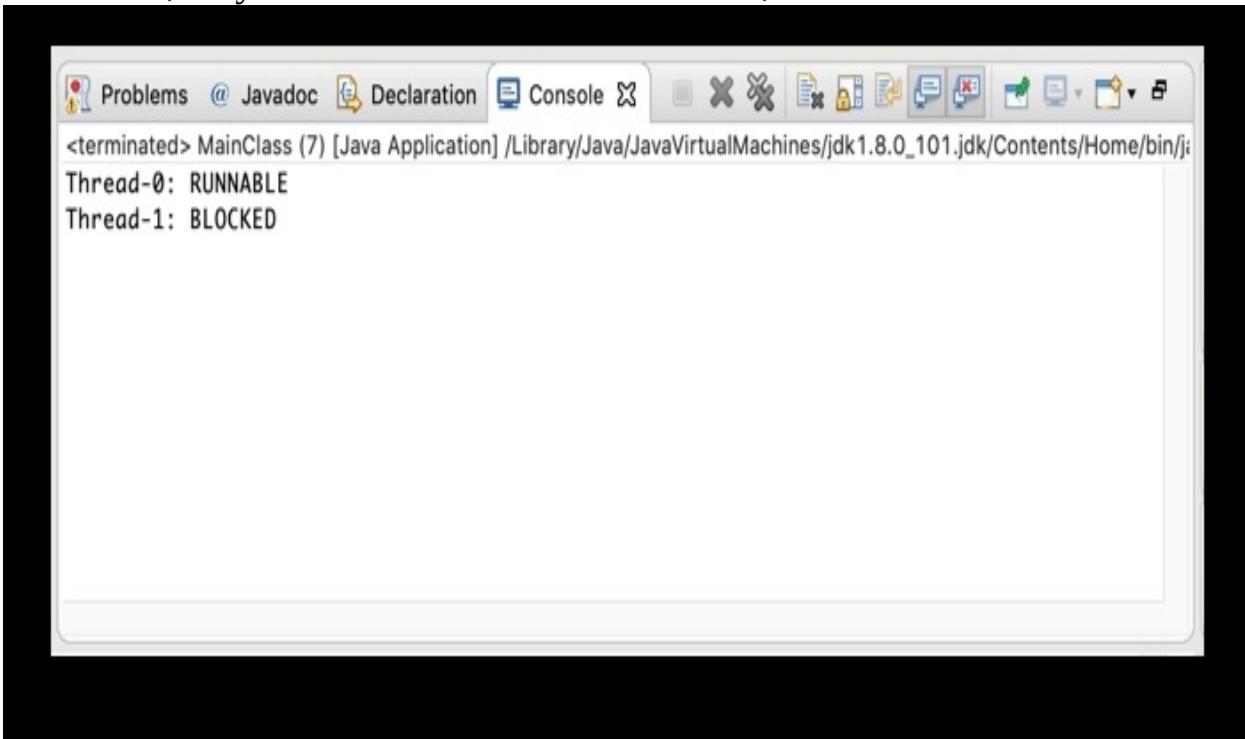
// Đều start() hết các đối tượng MyThread
// để xem Thread nào sẽ được vào commonResource() myThread1.start();
myThread2.start();

// In ra các trạng thái của chúng
System.out.println(myThread1.getName() + ": " + myThread1.getState());
System.out.println(myThread2.getName() + ": " + myThread2.getState());

}
}

```

As a result, only one Thread is now *RUNNABLE* , the rest will be *BLOCKED* .



4. WAITING

This state occurs when a thread has to wait for a thread to complete its task, for an indefinite amount of time. This state is different from the *BLOCKED* state above, the above is that Threads are locked by the system when they have common access to the same system resource. This state is between threads self-negotiating. *BLOCKEDs* are like vehicles blocked by a traffic police officer to

yield to other priority vehicles. Also *WAITING* self selfcompromise means of another, there is no need to regulate that which police. Since Threads will yield to each other, if a Thread makes a call to one of the following methods, it will "yield" and fall into this *WAITING* state , the methods are.

- *Object.wait ()*
- *Thread.join ()*
- *LockSupport.park ()*

Specifically, what the above methods are, invite you to the following lesson will clear. This we try to practice with the *join () method* . When a Thread calls the *join () method* of another Thread, it will have to wait for that other Thread to die before it can continue its remaining tasks.

Note that the methods I listed above have no parameters passed.
We will come to practice to better understand this state of Thread.

- Exercise Number 4

First, let's modify the *MyRunnable* class a bit as follows. public class MyRunnable implements Runnable {

```
@Override  
public void run() {  
    System.out.println("MyRunnable Start");  
    for (int i = 0; i < 100; i++) {  
        try {  
            Thread.sleep(100);  
        } catch (InterruptedException e) { // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }  
    System.out.println("MyRunnable End");  
}
```

The above code has nothing to do with putting Thread in *WAITING* state . This code just prints out the console that shows *MyRunnable* has just been "started" , and then does something heavy, like *milliseconds* only. Finally, that *MyRunnable* was “ *Ended* ” . repeating 100 times, each iteration will sleep 100

it will print out the console showing

Next we come to the *MyThread* class . In this *MyThread* , when launched, we deliberately declare and then launch *MyRunnable* . But when we just initialized *MyRunnable* , we called its *join () method* . This tells the system that, this *MyThread* will wait for *MyRunnable* to run out (end of *MyRunnable* 's *lifecycle*) before *MyThread* will continue running. You just code and launch it, for the next lesson I will talk more about this *join () method*.

Here is the code of *MyThread* . public class *MyThread* extends Thread {

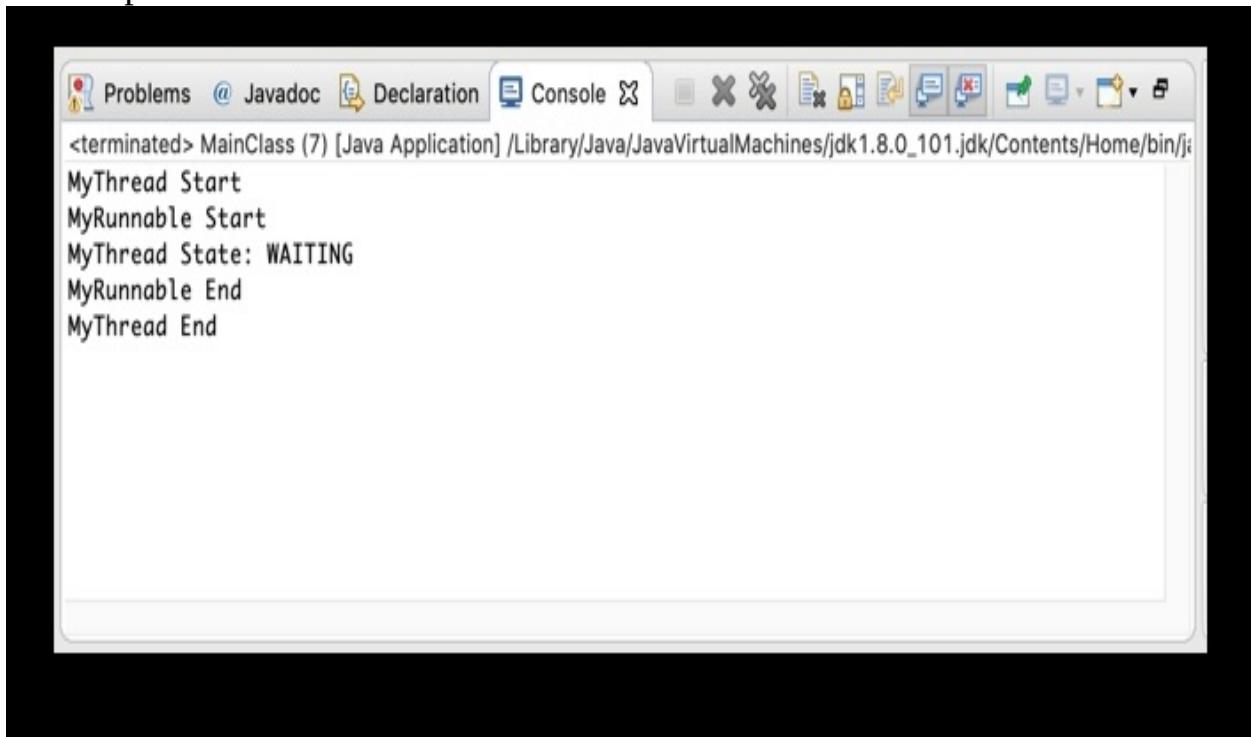
```
@Override  
public void run() {  
    System.out.println("MyThread Start");  
    Thread myRunnableThread = new Thread(new MyRunnable());  
    myRunnableThread.start();  
  
    try {  
        myRunnableThread.join();  
    } catch (InterruptedException e) { e.printStackTrace();  
    }  
  
    System.out.println("MyThread End");  
}
```

Then in the *main () method* you just need to launch *MyThread* , wait about *100 milliseconds*, then print the status of *MyThread* to the console to watch play. The reason to wait a bit to print the status of *MyThread* is because to make sure *MyThread* has enough time to start *MyRunnable* again, then the time that *MyThread* enters *WAITING* gives to *MyRunnable* too, you print too quickly, it is difficult to see the status. status of *MyThread* .

```
public static void main(String[] args) {  
    MyThread myThread = new MyThread();  
  
    myThread.start(); try {  
        Thread.sleep(100);  
        System.out.println("MyThread State: " + myThread.getState());  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }
```

}

The output to the console is as follows.



A screenshot of an IDE's Console tab. The window title is '<terminated> MainClass (7) [Java Application]'. The path is '/Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/ji'. The console output shows the following sequence of messages:

```
MyThread Start
MyRunnable Start
MyThread State: WAITING
MyRunnable End
MyThread End
```

5. TIMED_WAITING

Similar to *WAITING* above, but when the methods make one Thread "*yield*" to another thread to execute, pass the argument of the amount of time that the Thread passes. Those methods are.

- *Thread.sleep (long millis)*
- *Object.wait (int timeout)* or *Object.wait (int timeout, int nanos)*
- *Thread.join (long millis)*
- *LockSupport.parkNanos ()*
- *LockSupport.parkUtil ()*

Well, the *sleep () method* is familiar, isn't it. Now you understand, that somewhere in the Thread when calling this *Thread.sleep ()*, that Thread will fall into the state of *TIMED_WAITING* and "*yield*" to the other *Threads* to run in the specified milliseconds time.

However, we will also talk about these methods in the next lesson. Now let's see the code for the exercises.

- Exercise Number 5

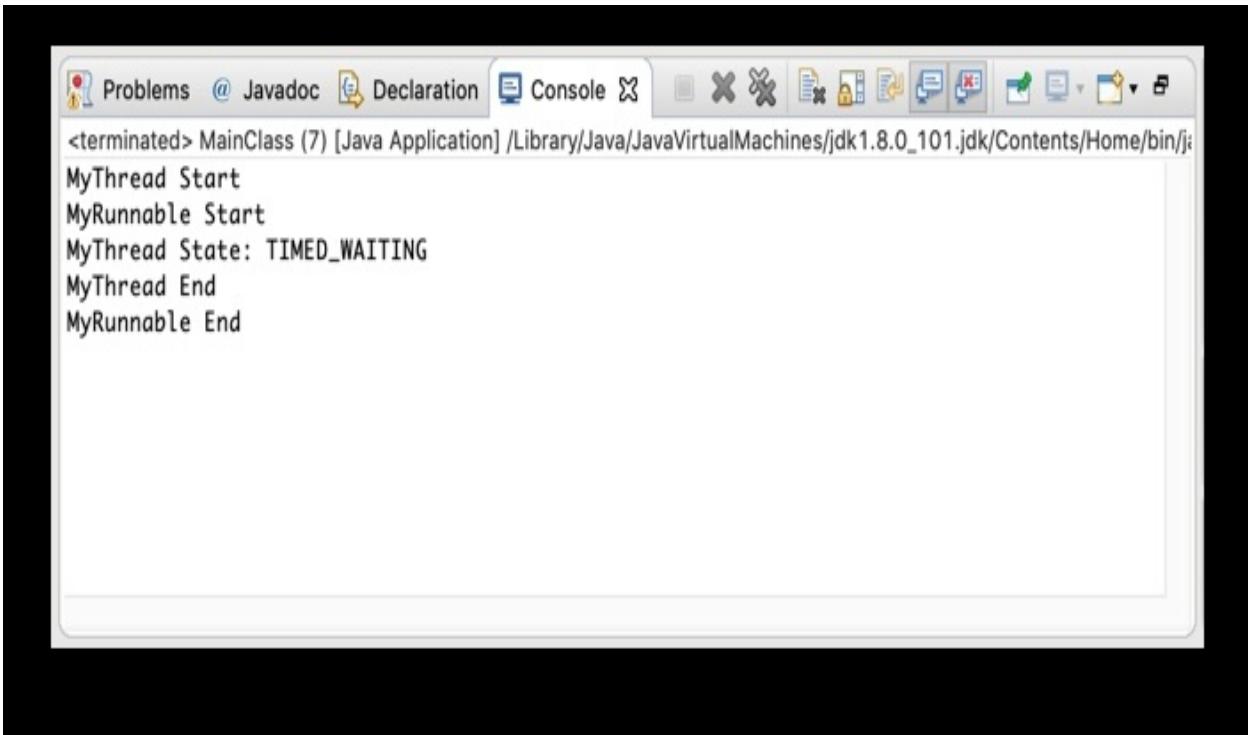
For this exercise, you will only need a little 4 above. Where *MyThread* starts *MyRunnable* and to *MyRunnable* is, now pass the value of milliseconds to this *join ()* method. It means that although *MyThread* has let *MyRunnable* run before, but only for a specified amount of time only, all the time that I run, touch anyone, touch.

public class MyThread extends Thread {
more editing compared to *Exercise* then calls *join ()* to yield

```
@Override  
public void run() {  
    System.out.println("MyThread Start");  
    Thread myRunnableThread = new Thread(new MyRunnable());  
    myRunnableThread.start();  
  
    try {  
        myRunnableThread.join(500);  
    } catch (InterruptedException e) { e.printStackTrace();  
    }  
  
    System.out.println("MyThread End");  
}
```

If you execute the program, be careful, after printing "*MyThread State: TIMED_WAITING*", *MyThread* will wait *milliseconds* remaining, and will print for *MyRunnable* for (less than) 500

" *MyThread End*" . The problem is that *MyRunnable* hasn't finished the loop yet, so it ends up printing " *MyRunnable End*" . Do you see the rhythm between threads?



6. TERMINATED

This state marks the end of Thread's lifecycle. Occurs when a thread ends all of its operations within its *run () method* , or has other unusual endings, such as falling into an *exception* .

- Exercise Number 6

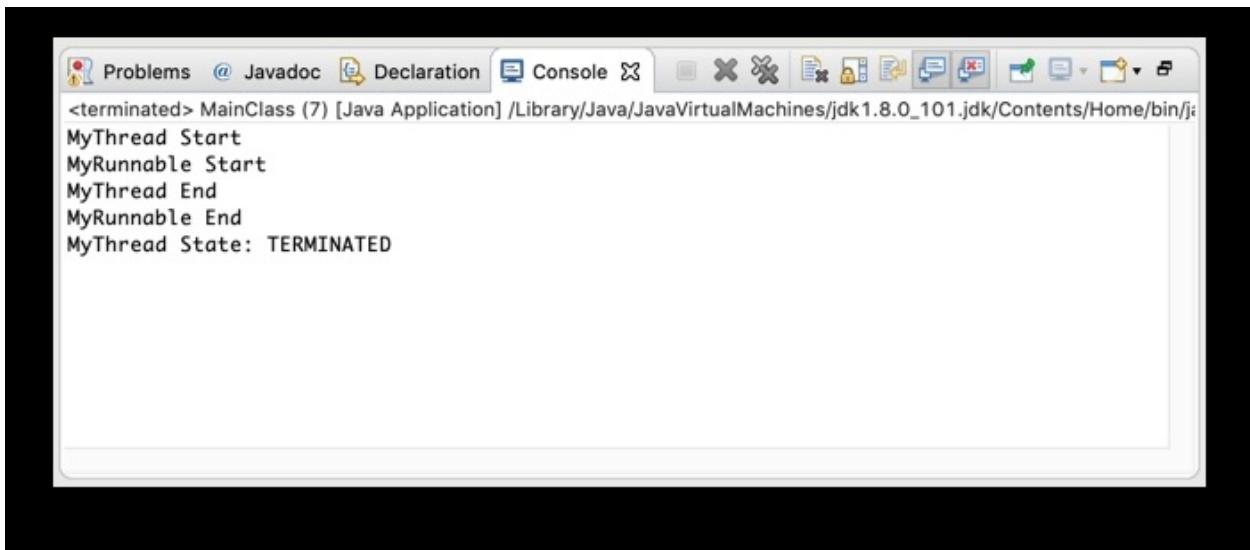
The code for this lesson is not much. You just get the code for *Exercise 5* above. Then in the *main () method* , you *sleep ()* for a while, in order to wait for *MyThread* to finish its task, then print its state. I give *20 seconds* of sleep , as follows.

```
public static void main(String[] args) {  
    MyThread myThread = new MyThread();  
    myThread.start();  
  
    try {  
        Thread.sleep(20000);  
        System.out.println("MyThread State: " + myThread.getState());  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```

```
}
```

```
}
```

The output to the console is as follows.



A screenshot of an IDE interface, specifically the Eclipse IDE, showing the 'Console' tab selected. The output window displays the following text:

```
<terminated> MainClass (7) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/ja
MyThread Start
MyRunnable Start
MyThread End
MyRunnable End
MyThread State: TERMINATED
```

Whew! We have just come across the knowledge of a Thread's lifecycle, through which we know how a Thread will experience its state during its lifetime. Surely you also know that not always a Thread goes through all the above states, there are *Threads* just *NEW* , *RUNNABLE* and *TERMINATED* . However, through this knowledge of the life cycle, you also have a rough idea of how threads synchronize, give way to each other to perform tasks, right? And knowledge about Thread is also quite a lot and no less interesting. See you in the following lessons.

Lesson 41: Synchronization (Part 1)

Remember the previous lesson I promised at the end of the lesson that this we will look at all the useful methods of Thread. But I realized that most of the cool methods that Thread brings revolve around how to help synchronize Threads in a process together.

So, to make it easier for the synthesis of the methods I promised, I decided to expose the knowledge of this *Synchronize Threads*. ?

As you approach the concept of *Synchronization* , you will have a better

understanding of useful methods within a Thread. This we will begin to familiarize ourselves with the early *Synchronization* concepts . Invite you to join the lesson.

I. What is Synchronization?

As you also know briefly in my about *Synchronization* , but not the data synchronization between offline devices with data in the cloud. Synchronization here is about how it works between threads. So what is it after all? ? I explain it clearly, through the sections about Thread, especially *Thread episode 3* Just now, you have definitely understood Thread, and you also know that there is a pretty headache in Thread, which is whether Thread is a great way for us to organize our tasks inside. the application is faster, smoother thanks to its parallel processing properties, then, this in turn leads to the risk that, at the same time, there may be more than one thread wanting to interfere with a user. Raw shared. We need to have a mechanism to help regulate so that at the same time, only one Thread has the right to use this shared resource, the other Threads have to wait their turn. This regulatory mechanism is called *Synchronization* (*Synchronized* in English , or *Synchronization*).

introduction lines above. This's lesson is

However, I would like to add one more issue of *Synchronization* . That is, if *Synchronization* is mentioned independently, then you can understand its function as I mentioned above.

Grateful if *Sync* is compared with *Real sync* (when the *synchronization* is *synchronous* , and *No synchronization* is *asynchronous*), the other back problems. *Synchronous* helps to organize the Thread in a certain order, after one thread is finished, the other can be executed, sequentially and smoothly.

Also *asynchronous* is disorganized ... disorganized, that is, we will not care which Thread finishes first, which thread ends later. And this *Synchronization series* we are only talking about *Synchronized* concept only(ie there is no comparison between *Synchronous* and *Asynchronous*).

II. When Will Synchronize Be Used?

As I said above, synchronization helps to adjust so that only one Thread at a time can be used to some shared resource. So when will this adjustment be required?

In fact, it is not always necessary for the resources within the application (which are files or objects) to be in sync with the system. Only resources that have a dispute, have a common thread between threads, leading to the risk of having one thread modifying the object's value, while another thread is also making modifications to this object. , leading to unnecessary "*misunderstandings*" in Threads, which may cause runtime crashes, ... then use *Synchronization* only.

A practical example is in the account management application of a bank. Suppose you are the builder of this account management application,? In your application there is an object that reads and writes a database of customer account balances. One day, a customer goes to an ATM to withdraw money from his account, assuming his account has *20 million VND* , he needs to withdraw *15 million VND*. . You also know that in order to withdraw money, the ATM (ie your application) must go through a balance check in that account, then this ATM receives the customer's withdrawal order and prepares to make the withdrawal (at this point your application object has just read the balance data, there is no balance correction). But at the same time, at home, the client's wife is also on the same account with her husband, proceeding? *20 million VND* In the husband's account through her account, after checking the balance in the account, the wife sees that there is still enough money (because the ATM has not done the deduction operation), the wife executes the order. ?transfers. And then what happened? Since your app's object sees the money remaining (both at the husband's ATM and the wife's transfer website), it does the subtraction? *15 million VND* in the database for the husband and *20 million VND* in the database for the wife. The husband and wife receive the money to be withdrawn, and, the wife's account also receives the money transferred. In short, the bank will lose money (*15 million VND*), you will be fired.

Now, now, I'm just assuming, sure, after reading this series on *Synchronization* , you can completely avoid mistakes that can happen in the future like for example. To make it easier to understand, let's redo this situation into a small? Project as follows.

- *Example Of Unynchronized Bank Withdrawal*

In this example, I invite you to build an application that simulates withdrawing money from the bank as above.

We first build an object that holds the customer's account balance information. This object is carefully constructed by you, it can check the account balance

before allowing withdrawal (balance is set up initially at *20 million*). After checking the balance and seeing the withdrawal allowed, it will deduct the balance in the account. Let's say that checking account balances and updating new balances back into the database take *2 seconds* for each operation. All are simulated through the following class code. This class I named *BankAccount*.

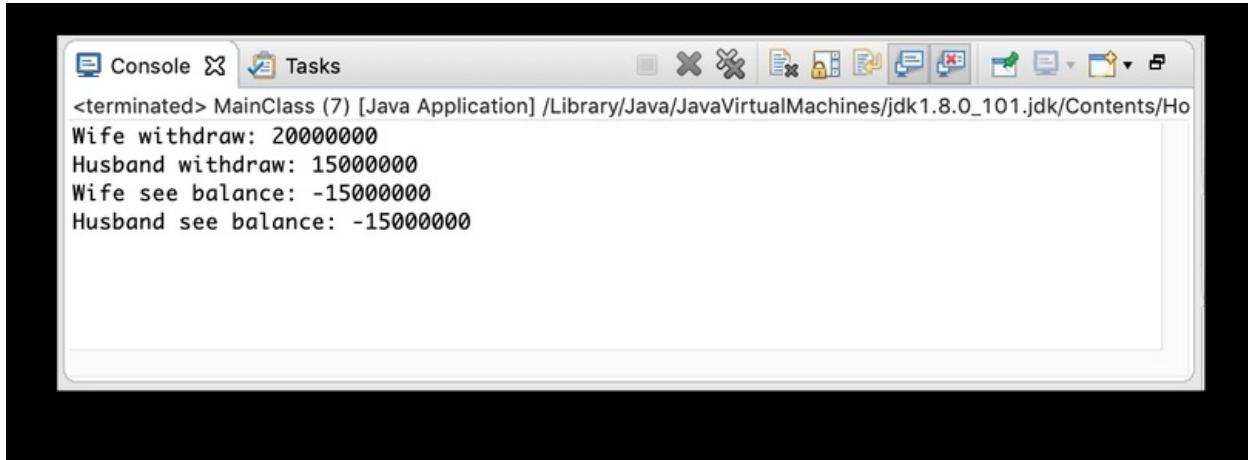
```
public class BankAccount {  
    long amount = 20000000; // The amount in the account  
  
    public boolean checkAccountBalance (long withdrawAmount) { // Simulate  
        time to read the database and check money try {  
  
            Thread.sleep (2000);  
        } catch (InterruptedException e) {  
            e.printStackTrace ();  
        }  
  
        if (withdrawAmount <= amount) { // Allow to withdraw money return true;  
        }  
        // No withdrawal allowed return false;  
    }  
  
    public void withdraw (String threadName, long withdrawAmount) { // Print  
        withdrawal information  
        System.out.println (threadName + "withdraw:" + withdrawAmount);  
  
        if (checkAccountBalance (withdrawAmount)) { // Simulate withdrawal time and  
            // update the remaining amount into the database try {  
  
                Thread.sleep (2000);  
            } catch (InterruptedException e) {  
                e.printStackTrace ();  
            }  
  
            amount -= withdrawAmount; }  
            // Print out the account balance  
            System.out.println (threadName + "see balance:" + amount); }  
    }
```

The above code is easy to understand, right? You already know that this *BankAccount* is a shared resource. And already a shared resource, you should build a withdrawal Thread as follows. This thread will allow to pass to this shared resource, and pass the amount to withdraw, then it will call the withdrawal method of that resource. I named this withdrawal Thread *WithdrawThread*.

```
public class WithdrawThread extends Thread {  
  
    String threadName = ""; long withdrawAmount = 0; BankAccount  
    bankAccount;  
  
    public WithdrawThread(String threadName, BankAccount bankAccount, long  
    withdrawAmount) {  
        this.threadName = threadName;  
        this.bankAccount = bankAccount;  
        this.withdrawAmount = withdrawAmount;  
  
    }  
    @Override  
    public void run() {  
        bankAccount.withdraw(threadName, withdrawAmount); }  
    }  
  
Finally, the code for the main () method is quite simple, we just need to declare 2  
Thread to withdraw money and let them use the same resource bankAccount  
only.  
public static void main(String[] args) {  
    BankAccount bankAccount = new BankAccount();  
    // Người chồng rút 15 triệu  
    WithdrawThread husbandThread = new WithdrawThread("Husband",  
    bankAccount, 15000000);  
    husbandThread.start();  
  
    // Người vợ rút hết tiền (20 triệu)  
    WithdrawThread wifeThread = new WithdrawThread("Wife", bankAccount,  
    20000000); wifeThread.start();  
  
}
```

And when executing this program, you will see the result as shown below. This result is the process where two Thread *husbandThread* and *wifeThread* together

check the account and find it possible to withdraw, then they both execute the withdrawal order, and the result both see successful execution . But... its balance is negative (means the bank loses money).



The screenshot shows a Java application running in an IDE. The console tab displays the following output:

```
<terminated> MainClass (7) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Ho
Wife withdraw: 20000000
Husband withdraw: 15000000
Wife see balance: -15000000
Husband see balance: -15000000
```

This output indicates a race condition where both threads attempt to withdraw money from the same account simultaneously. Both threads successfully withdraw their respective amounts (20,000,000 and 15,000,000), but both also print their final balance as -15,000,000, which is incorrect.

We just stop at the above example, to the next lesson, when we begin to know the specifics of the *Synchronization* methods , we will know how the above bank is not lost. money, and maybe keep the life, no, the job for you.

III. Methods of Synchronization

In this series of synchronization lessons, I will divide them into 2 ways, which are also the two most for us to easily reach and remember.

The first is called a Mutual Exclusive . Can be understood as *Mutual Exclusion* . This way the system will prioritize one thread and help prevent other threads from the risk of conflict. Because of this feature of the mechanism, it reminds us of a drastic and drastic intervention of the system, so the name "*eliminates*" . This way will be encapsulated in the next lesson.

The second is called Cooperation . It can mean *Collaboration together* . This way the Threads will shake hands with each other, and together regulate the priority order to avoid conflicts by themselves. This way will be discussed in the next lesson on the above exclusion method.

In fact, in the previous lesson, you have also been briefly familiar with these two synchronous ways. Remember it. *This link* is the *Mutual Exclusive* way , and *this link* is *Cooperation* .

Done, light lesson is not it. The knowledge of the lesson focuses only on the introduction of the concept of *Synchronization* and the two major ways of *Synchronization* that we will look at them in the next sections.

Lesson 42: Synchronization Part 2

So after finishing the opening lesson about Syncing the *day before* , I said that there will be two ways to synchronize threads together. These synchronizations all have a common goal of restricting Threads from accessing the same shared resource. And this's lesson I will detail the first of the two ways above, this method is called *Mutual Exclusive* . After these synchronization lessons, you will learn how to avoid system resource conflicts when working with *Multithread* , and also know when to use which method to synchronize.

Invite you to join the lesson.

I. What is Mutual Exclusive Sync?

Certainly *Mutual Exclusive Synchronization* is a method of synchronizing Thread, helping Threads be synchronized so that they do not interfere with shared resources at the same time. So why is it called *Mutual Exclusive* ? *Exclude* here means *Block* , which means the system will block Threads that call the same shared resource, and only allow one Thread to use this resource. Those blocked threads will have to wait until they are stopped before being able to use that resource. In short, *Exclude* here is understood as *Prevent* , not to *Abort* Thread.

So how will the system perform that elimination, or prevention? For the easiest visualization, let's take a practical example below (this is the easiest to understand example of a *Mutual Exclusive* that I see many documents used). For example, in a certain conference there were many speakers sitting together, they all talked about a topic of how to learn Java as best as possible (this topic was made by me). The same problem as with the Thread we're talking about is, there's only one topic, each speaker has an opinion, and everyone scrambles to raise their opinion. As a result, the listener will receive mixed information, no one can understand what the conference is bringing.



If you consider each speaker a Thread, and the topic they are talking about is shared resources. The speakers themselves are the factors that make the topic so cheesy. To solve this problem, you come up with a mechanism, this mechanism called *Mutual Exclusive*. The idea of the mechanism is that speakers have to take over their own right to speak, to exclude other speakers' rights to speak, forcing other speakers to listen until the speaker is speaking. That is the story. Oh so what to do, you can't step in and specify who will talk and who will have to listen, because it would take too much work. No, you don't. You give them a microphone. Bum! The problem has been resolved. With a microphone placed on the table, it is the speaker who receives the microphone towards him, the speaker has the right to speak, the others listen until the speaker is handed over to the microphone. The listening speakers have the right to subscribe to speak to a list, so that when the other speaker finishes speaking, the next person on the list will have access to the microphone.



The above example clearly shows the method to perform *Mutual Exclusive* sync already. We only need to consider when applying the above example to our knowledge of Thread Synchronization, how the system will do, invites you to the next section.

II. How is Mutual Exclusive Sync?

The mechanism that the speaker is only allowed to say when there is a microphone, when applied to Thread synchronization, it is called with another name *Monitor & Lock*, or many documents called *Monitor Lock*. I don't have to understand that *Monitor* is the screen and *Lock* is a lock. This mechanism is understood that the microphone, or other shared documents, will be protected by an object called the *Monitor*. For each speaker (or Thread) that wants to use the microphone (or shared resource), must register through *Monitor* to get a *Lock*. Each *Monitor* will have only one *Lock*. Which thread can get the above *Lock* That *Monitor*, that Thread is allowed to use the shared resource, until the Thread

finishes using the resource and returns the *Lock* to *Monitor*, this *Lock* will be passed to the next Thread in the waiting list in *Monitor*, so that the next Thread has a chance to use it, and *Lock* the resource. Just like that the *Lock* is passed on for all threads waiting in *Monitor*.

The mechanism is like that, it is not too difficult to understand right. So how to apply the *Monitor Lock* to our code, we invite you to the next section.

III. Keyword synchronized

We are getting used to a synchronization approach called *Mutual Exclusive*. We know that the mechanism by which the system performs synchronously is called the *Monitor Lock*. And in order to call the system using this mechanism to synchronize, we have to get used to using a new keyword in Java, this keyword is named *synchronized*.

This means, when we want something to be protected by *Monitor*, then put the *synchronized* keyword inside. Using the *synchronized* keyword inside an object, I will talk about it in the specific sections below. Your job should now understand that, when an object has the *synchronized* keyword internally, it will be managed by the system in a *Monitor*. Each object will have a separate management *Monitor*.

And so, as you know, Threads want to use *synchronized* methods inside that object, it must have a *Lock*. And when a *Monitor* is given of the object it manages *Lock* to a certain Thread, it must wait for that Thread to return the *Lock* before another Thread can use these *synchronized* methods. And so your Synchronization problem is solved.

Basically, using *synchronized* is not difficult. First, you can understand that *synchronized* can be declared at the *method level in the class*, or at the *block level* within the method.

We will conduct survey of each *synchronized* type in the following specific items.

1. Use *synchronized* For Method

When declaring a method, if you want to synchronize on this method, add the *synchronized* keyword as shown in the code below.

```
public synchronized void withdraw() { //...}
```

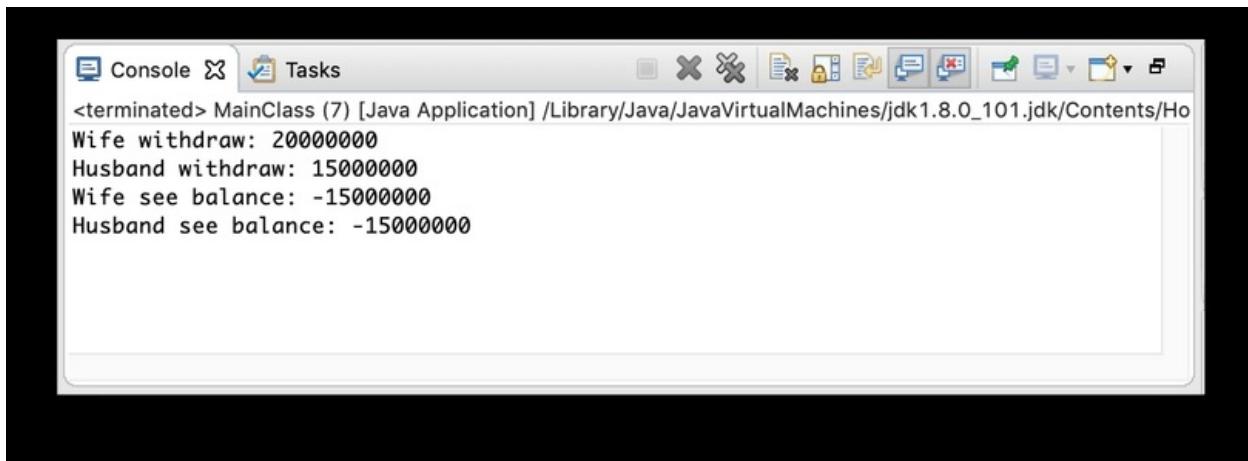
```
// ...
```

```
}
```

To make it easier to understand, let's go to the practice. • Practice # 1

In this exercise we will take back the *bank withdrawal example* in the previous lesson.

I summarize a bit in this example. In the previous example you built a class *BankAccount*. This class contains two methods *checkAccountBalance ()* and *withdraw ()*, which in turn are methods of checking balances and withdrawing money from the bank if it is still enough to withdraw. Then we declare two Thread is *husbandThread* and *wifeThread* then proceed with withdrawal, the results obtained in the previous lesson the following day.



The lines on the console indicate that the two Threads want to withdraw money, and the result after the withdrawal on both Threads is seen as negative, indicating that the balance check has made an error, due to both checks of balance. Both Threads find it available, and both perform a withdrawal with the total amount exceeding the allowed balance.

Thus we need to synchronize these Threads, namely, interfering with the shared resource *BankAccount*. Make *BankAccount* object protected by *Monitor*. And then the Thread wants to use the methods of this object, they must request *Lock*. So according to the lesson, we just need to add the *synchronized* keyword to *BankAccount* as follows (I have changed the code to print the console to make it clearer than the previous lesson).

```
public class BankAccount {
```

```

long amount = 20000000; // The amount in the account
public synchronized boolean checkAccountBalance (long withdrawAmount) { // 
Simulate time to read the database and check money
try {

    Thread.sleep (2000);
} catch (InterruptedException e) {
    e.printStackTrace ();
}

if (withdrawAmount <= amount) { // Allow to withdraw money return true;
}

// No withdrawal allowed return false;
}

public synchronized void withdraw (String threadName, long withdrawAmount)
{ // Print withdrawal information
System.out.println (threadName + "check:" + withdrawAmount);

if (checkAccountBalance (withdrawAmount)) { // Simulate withdrawal time and
// update the remaining amount into the database try {

    Thread.sleep (2000);
} catch (InterruptedException e) {
    e.printStackTrace ();
}

amount -= withdrawAmount;
System.out.println (threadName + "withdraw successful:" + withdrawAmount);
} else {
    System.out.println (threadName + "withdraw error!");
}
// Print out the account balance
System.out.println (threadName + "see balance:" + amount); }
}

When it comes to performance, compare the results.

```

The husband want to withdraw 15mil The husband withdraw sucessful 15mil The balance is 5mil
The wife want to withdraw 20mil

```
Console X | 
<terminated> Main (97) [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (Sep 2, 2020 11:08:22 AM - 11:08:23 AM)
Husband check 15000000
Husband withdraw scucessful 15000000
Husband see balance 5000000
Wife check 20000000
Wife withdraw error!
Wife see balance 5000000
```

The wife can't do that

The balance is still 5mil



You see, the result of this synchronization is, *husbandThread* requests a withdrawal first, it will be issued a *Lock* first, until *husbandThread* finishes withdrawing money , *wifeThread* can begin to execute and not be contested. accept anything.

2. Use synchronized For Method Inside Block

At this point, you have partly understood how the *synchronized* keyword works , right? To make it clearer, when you put the *synchronized* keyword in one or more methods inside the class. Then the object of that class will be managed by *Monitor* , once a Thread registers to use one of the methods with the *synchronized* keyword , *Monitor* grants a *Lock* to that thread until it completes those methods..

So there are times when you do not need to ask for *Lock* for the entire method. If you only need one part of the method it is protected by *Monitor*. Then, apply the *synchronized* method for the block of this item.

You can refer to the syntax of *synchronized synchronized* to the statement block inside the method as follows.

```
synchronized (object) {  
// Content of block  
}
```

Syntax is not too difficult, you just need to consider parameters *doi_tuong* pass to the block *synchronized* only. This parameter tells the system which object should only be managed by its *Monitor* for synchronization. For better understanding invite you to practice.

- Practice # 2

We will retrieve the code of the *BankAccount* class in *Exercise 1* above. But for now we just need to synchronize one block of code highlighted later. You see that apart from wrapping this *synchronized* block in familiar lines of code, and removing *synchronized items* from the methods as in Practice # 1, everything doesn't change.

```
public class BankAccount {  
long amount = 20000000; // The amount in the account  
  
public boolean checkAccountBalance (long withDrawAmount) { // Simulate  
time to read the database and check money try {  
  
Thread.sleep (2000);  
} catch (InterruptedException e) {  
e.printStackTrace ();  
}  
  
if (withDrawAmount <= amount) { // Allow to withdraw money return true;  
}  
// No withdrawal allowed return false;  
}
```

```

public void withdraw (String threadName, long withdrawAmount) { // Print
withdrawal information
System.out.println (threadName + "check:" + withdrawAmount);

synchronized (this) {
if (checkAccountBalance (withdrawAmount)) { // Simulate withdrawal time and
// update the remaining amount into the database

try {

Thread.sleep (2000);
} catch (InterruptedException e) {
e.printStackTrace ();
}

amount -= withdrawAmount;
System.out.println (threadName + "withdraw successful:" + withdrawAmount);
} else {
System.out.println (threadName + "withdraw error!");
}
}

// Print out the account balance
System.out.println (threadName + "see balance:" + amount); }
}

```

With the above code, I synchronize only a small block of commands. Screen prints are out of sync, and when running the application, the results are slightly different. However, the application still runs correctly.

As mentioned in the above syntax, passing *this* to the *synchronized* block tells *Monitor* to take protection on this object, but only for protection in the block.



If with practice number 1, the synchronization is on both checking and withdrawing methods, so when the husband goes in first, the system will check that the husband has completed all the operations to serve. for the wife. Thus,

with the system of Practice No. 1, when the wife comes to use the system, it will take a long time to see the response system, because the husband still has to wait for the husband to finish. As for this exercise, you find that the system will respond to both (print available balance on the screen) because the synchronous lines of code have not yet reached. Until the husband officially enters the method of checking money, the new system performs *Mutual Exclusive* to the wife. And as you can see on the console above.

So we have looked at the first of the ways to synchronize Thread. The usage of this *synchronized* keyword is quite simple, and I see it is used a lot for Multithread conflict avoidance cases.

Lesson 43: Synchronization Part 3 Sync Cooperation

It's been a while for this sequel Java lesson. This can be considered as a Java lesson to start the new year, but is a topic "*owed*" from the old year. Maybe because the long wait will make you forget a bit. Again, we are talking about Thread *Synchronization methods* in Java programming. We find a way to make Threads "*freely free*" in the execution of parallel tasks, can know to follow certain order rules when they have common use to objects. statues, or we call resources. The previous lesson was one way, this we come to the second. Invite you to the lesson.

I. What is Sync Cooperation?

Synchronization Cooperation , or some document called *Inter-Thread Communication* , is the purpose of this synchronization method unintentionally to avoid conflicts from threads when they are simultaneously using the same object, or system resources. So why is it called *Cooperation* ? Unlike the previous lesson that outlined the *Mutual Exclusive* synchronization method, this method of the previous lesson creates an exclusion mechanism, which helps Threads that use resources first will be prioritized, Threads used later will be excluded. and have to wait. This's method is the exact opposite. There is no "*first come first*" anymore. But they have "*cooperation*" with each other (*Cooperation*), collaborating in a spirit that one Thread can completely "*yield*" to another thread to use the resource it "*won*" first, so that when another Thread has finished using that resource, that Thread must "*wake*" it up in order for it to continue working on that resource.

II. How to Sync Cooperation?

As you also know. The core of this's synchronous approach is that threads will have to adjust and yield to each other in resource usage. Therefore, we must become familiar with the methods involved in modifications and yield, instead of just one keyword like the previous lesson. We are talking *wait ()*, *notify ()*, and *notifyAll ()* methods .

Before let's learn the meaning and usage of these 3 methods, I want to repeat a little about the *Monitor & Lock* mechanism mentioned in the previous lesson. Because the synchronization of this's lesson is not out of the use of this *Monitor* and *Lock* .

As mentioned in the previous lesson, each object in the system has a managed *Monitor* . Each *Monitor* has only one *Lock* . When the thread that wants to use the object, it must be registered through the *Monitor* of the object, *the Monitor* will hand *Lock* on the Thread that object. The *synchronized* keyword on the to be entitled to use the object that you are familiar with tells *Monitor* that if you give *Lock* to some thread, the other thread will have to stay in *Monitor and wait* until *Lock* is returned.

So still with this *Monitor & Lock* mechanism , the lesson this will be like. We come together with the methods that we just mentioned above. Note that these methods are built in the *Object* superclass , which means that all objects in Java have these methods.

1. *wait ()*

This method, when called, will cause the Thread holding the *Lock* on the object to return this *Lock* to the *Monitor* of that object. At the same time that thread falls asleep, waiting for another thread to *wake up* by one of the two methods below.

2. *notify ()*

As mentioned above, this method helps to "wake" Thread that has fallen asleep by the *wait () method* .

3. *notifyAll ()*

This method is more extensible to *notify ()* . It *wakes up* all threads that have

called `wait()` within this object.

As you can see, the core idea in using the synchronous methods of this's lesson is "*yield*". A Thread has entered the *Monitor* of an object first, get the *Lock* of that object, but because of some real situation, that Thread still hasn't used this object yet. It proceeds `wait()` to let a certain thread to use this object first, then wake it up, so that it can continue the current work in progress.

To better understand how to use methods in synchronization this, let's go to the following exercise.

III. Practice Solving Withdrawals From Banks

Together we continue to build banking related operations. You can review the *previous exercise* to review the old synchronization patterns, and compare it with the script for this's exercise.

The day before, you have built up a "*perfect*" algorithm , where both the husband and the wife perform withdrawals on the same account. You have made the application distinguish between who withdraw first, who withdraws later, to ensure the balance check sequentially, helping the bank not suffer "*loss*" when there is a situation like this withdrawal at the same time. And I think the conflict-avoiding method of the previous lesson is completely suitable for such a scenario.

Therefore, this we do not use this scenario anymore. We change a little bit. Suppose the bank you are working with has a new service, which is to assist the user to place withdrawal orders as soon as the account balance is sufficient for that customer to withdraw. This means that, if, while the client places a withdrawal order, and the bank account is sufficient for this operation, the client will receive the money immediately, but if not enough, it will wait until the account is just have enough money, then execute the withdrawal order. The requirements of this second part are a bit complicated. If you do not read this's lesson, you can build for the application a function to regularly check the client's account, check once every minute for example, as soon as there is enough money to withdraw, the order will be executed. withdraw immediately. The idea of periodic testing seems to be correct, but not good, it will slow down the system if you have too many customers.

And the real situation is, with a husband, his account is *5 million VND* . The wife wants to withdraw *10 million VND* but is not enough, she uses the withdrawal service as soon as the account is enough as mentioned above. One day, the husband deposited *5 million VND* into his account . Eligible. Application makes a withdrawal with the wife instantly.

To do this, we will construct the *BankAccount* class as follows. As you know, *BankAccount* is the layer that manages balance information. Threads of the husband or wife use this class to change the account number. The *BankAcount* class has been built in from the previous lesson with the methods *checkAccountBalance ()* and *withdraw ()* for checking balances and withdrawals in parallel. In this lesson we will build two more methods which are *withdrawWhenBalanceEnough ()* and *deposit ()* for the request of the new service that I just presented above. *BankAccount* class :

```
public class BankAccount extends Object {  
    long amount = 5000000; // The amount in the account  
  
    public synchronized boolean checkAccountBalance (long withDrawAmount) { //  
        Like the code of the previous lesson, you copy / paste it yourself, // this lesson I  
        do not display again  
  
    }  
    public synchronized void withdraw (String threadName, long withdrawAmount)  
    { // Like the code of the previous lesson, you copy / paste it yourself, // this  
        lesson I do not display again  
  
    }  
  
    public synchronized void withdrawWhenBalanceEnough (String threadName,  
    long withdrawAmount) { // Print withdrawal information  
        System.out.println (threadName + "check:" + withdrawAmount);  
  
        while (! checkAccountBalance (withdrawAmount)) {  
            // If there is not enough money, then wait until there is enough money, then  
            withdraw  
            System.out.println (threadName + "wait for balance enough");  
            try {  
                wait ();  
            }  
        }  
    }  
}
```

```

} catch (InterruptedException e) {
// TODO Auto-generated catch block
e.printStackTrace ();

}

}

// Having enough money, or no longer waiting, it is allowed to withdraw //
Simulate withdrawal time and
// update the remaining amount into the database
try {

Thread.sleep (2000);
} catch (InterruptedException e) {
e.printStackTrace (); }
amount -= withdrawAmount;
System.out.println (threadName + "withdraw successful:" + withdrawAmount);
}

public synchronized void deposit (String threadName, long depositAmount) { //
Print the depositer's information
System.out.println (threadName + "deposit:" + depositAmount);

// Simulate recharge time and
// update the new amount into the database try {

Thread.sleep (2000);
} catch (InterruptedException e) {
e.printStackTrace ();
}

amount += depositAmount;
// Wake up the sleeping object and wait for money to withdraw
notify ();

}
}

```

Notice the *wait ()* and *notify ()* methods used in the *BankAccount* methods . First, you should know that the synchronous methods of this's lesson must be kept

inside *synchronized* block to avoid collisions first. So the *withdrawWhenBalanceEnough ()* and *deposit ()* methods are both *synchronized* methods . Please review the *previous lesson* if you do not understand what the keyword *synchroziered* is used for.

Go back to *BankAccount* . The *withdrawWhenBalanceEnough () method* has just entered has checked balance. If the balance is not enough, it immediately meets the command *wait ()* . With that said, this command causes the Thread holding the current *Lock* to return the *Lock* and fall asleep, waiting for some other Thread to wake up. The *while* loop in checking the balance in this paragraph helps the Thread, when alive, to still have to check the balance again. If the wake up balance is sufficient, the withdrawal will be executed in the lines of code below it. The balance is not enough, *wait ()* and sleep again. Do you understand?

And the *deposit () method* at *BankAccount* will be the method of recharging the account. After the recharge is complete, this method keeps calling *notify ()* to wake up some sleeping thread and wait to be withdrawn if any. And actually *notify ()* at *deposit ()* does not know which thread is sleeping and waiting to wake up, so if you are sure, you just call *notifyAll ()* to wake up all threads called *wait ()* side. in *BankAccount* is fine.

Then, to save the trouble, we build 2 Threads, a Thread to withdraw if enough, and a Thread to recharge.

Thread withdraws as follows.

```
public class WithdrawThread extends Thread {
```

```
    String threadName = ""; long withdrawAmount = 0; BankAccount  
    bankAccount;
```

```
    public WithdrawThread(String threadName, BankAccount bankAccount , long  
    withdrawAmount) { this.threadName = threadName;  
    this.bankAccount = bankAccount;  
    this.withdrawAmount = withdrawAmount;
```

```
}
```

```
    @Override public void run() {
```

```
        bankAccount.withdrawWhenBalanceEnough(threadName, withdrawAmount);
```

```
}
```

```
}
```

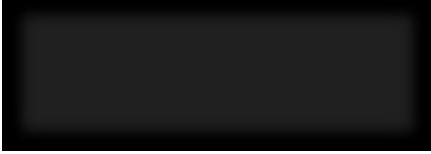
Thread withdrawal is easy to understand, right? Nothing new. And here is the recharge Thread.

```
public class DepositThread extends Thread {  
  
    String threadName = ""; long depositAmount = 0; BankAccount bankAccount;  
  
    public DepositThread(String threadName, BankAccount bankAccount , long  
depositAmount) { this.threadName = threadName;  
this.bankAccount = bankAccount;  
this.depositAmount = depositAmount;  
  
}  
@Override  
public void run() {  
bankAccount.deposit(threadName, depositAmount);  
  
}  
}
```

Threads withdraw and recharge is no different, we just split 2 Threads for a while to the *main () method* easy to see. You can merge both of these 2 threads into 1 still. You try it.

```
The method main () we will call 2 Thread as follows.  
public class MainClass {  
public static void main (String [] args) {  
BankAccount bankAccount = new BankAccount ();  
  
// The wife wants to withdraw 10 million VND  
// (pay attention when the money is not enough to withdraw)  
WithdrawThread wifeThread = new WithdrawThread ("Wife", bankAccount,  
10000000); wifeThread.start ();  
  
// Husband and husband deposit 5 million VND  
DepositThread husbandThread = new DepositThread ("Husband", bankAccount,  
5000000);  
  
husbandThread.start ();  
}  
}
```

At this point, you can execute the program to see how the results are printed to the console.



We have just completed the 3rd of the Thread Synchronization series of knowledge. But it is the 6th knowledge in Thread knowledge series already. You can see the importance of Thread. We conclude our Thread Synchronization series here. But it's not the end of Thread knowledge chain yet. And next lesson we will come to a new knowledge about Thread.

Lesson 44: Deadlock

By the end of the previous lesson, we are done with Thread Synchronization. You saw the very powerful role of the *synchronized* keyword in making sure there is no conflict over shared resources. Indeed *synchronized* is very good, but if you misuse it in the wrong place, you will have a situation that this's lesson is aimed at. That situation is called *Deadlock*.

I. What is Deadlock?

At first glance, the name makes us think of *some "Dead"* (Dead) ?!?! It can be said that this's lesson is not outside of death. More specifically, I am talking about the death of our application, which is caused by threads in your program that "*wait*" for each other to die!

If the "*death*" sounds too scary, then I invite you to come with the following two situations, the first one will take fun examples from the reality, the following scenario will go into detail in programming Deadlock. What is it.

1. Understanding Deadlock Through Realistic Examples

Although this is a situation that does not lead to the "*waiting to die*" as in programming, but it also makes the litigants not know how to use it.

The picture below shows a policeman holding a robber. The policeman wanted the other robber to hand over the hostages first before he released the robber he was holding. Meanwhile, the other robber definitely did not return the hostage,

forcing the policeman to release his accomplices first.



Simulate a deadlock possibility in reality So, each side in this situation holds their own hostages, and neither side will return hostages to the other side. This situation is clearly difficult to reach in the short term. *Deadlock* when this happened.

2. Deadlock In Programming

In programming, the *Deadlock* situation is similar to the fun reality example above. Let's say *Police* and *Robbery* are each Thread. Then the *gang of robbers* and the *Hostages* are the resources. The *Police* Thread is holding the *Robber's accomplices* through the *synchronized* keyword , but the *Police* Thread really wants to keep *Hostages* . Which *I believe* are being held by Thread *Pirates* also by keywords *synchrozierd* , while *Pirates* also want to get the *posse robbery* . In programming, then *Deadlock* will also happen.

II. Application Construction Practices Causing Deadlock

If the above are practical and theoretical situations for you to better understand *Deadlock* . Then now I invite you to build a real application, capable of causing *Deadlock* .

We will still come to the bank application building scenario like the Thread synchronization lessons that you are already familiar with. Suppose this the bank boss comes to you and tells you to build more functions of transfer between accounts. After the construction function is completed, there are two married couples in a family. The husband has his own account, and the wife also has a separate account at the same bank. One day, because he did not understand each other, the husband who was not in his account transferred to his wife *3 million VND*, at the same time, the wife also transferred to her husband *2 million VND from her account*. . The irony problem is that these 2 people perform money order at the same time. And strangely enough, the application was suspended, which meant that the couple waited all the time and the money order still failed. Why, let's take a look at the code you wrote.

Assuming your *BankAccount* class has built-in withdraw and deposit methods built from previous lessons. Here I write shorter than the previous lesson, ignore the balance check and simulate the time to withdraw / recharge for this class as concisely as possible.

```
public class BankAccount extends Object {  
    long amount = 5000000; // The amount in the account  
    String accountName = "";  
    public BankAccount (String accountName) { this.accountName = accountName;  
    }  
  
    public synchronized void withdraw (long withdrawAmount) { // Print out the  
        status to start subtracting money System.out.println (accountName +  
        "withdrawing ...");  
  
        // Deduction  
        amount -= withdrawAmount; }  
  
    public synchronized void deposit (long depositAmount) { // Print out the status  
        of starting to recharge System.out.println (accountName + "depositting ...");  
  
        // Recharge  
        amount += depositAmount; }  
}
```

And now you build more transfer methods for this class. You name it *transferTo ()*. Due to being too careful, you add the *synchronized* blocks in this method. The complete code of the *BankAccount* class will look like this.

```

public class BankAccount extends Object {
    long amount = 5000000; // The amount in the account
    String accountName = "";

    public BankAccount (String accountName) { this.accountName = accountName;
    }

    public synchronized void withdraw (long withdrawAmount) { // Print out the
        status to start subtracting money
        System.out.println (accountName +
        "withdrawing ...");

        // Deduction
        amount - = withdrawAmount; }

    public synchronized void deposit (long depositAmount) { // Print out the status
        of starting to recharge
        System.out.println (accountName + "depositting ...");

        // Recharge
        amount + = depositAmount; }

    public void transferTo (BankAccount toAccount, long transferAmount) {
        synchronized (this) {
            // Withdraw money from this account
            this.withdraw (transferAmount);

            synchronized (toAccount) {
                // Deposit toAccount
                toAccount.deposit (transferAmount);

            }
        }
    }
}

// Print account balance at the end of the transfer
System.out.println ("The amount of" + accountName + "is:" + amount); }
```

In the *main () method*, just call the transfer instructions as follows.

```

public static void main (String [] args) {
    // Declare the husband and wife's account
    BankAccount husbandAccount = new BankAccount ("Husband's Account");
    BankAccount wifeAccount = new BankAccount ("Wife's Account");
```

```
// The husband wants to transfer 3 million from the photo's account to the wife's
account
Thread husbandThread = new Thread () {
@Override
public void run () {
husbandAccount.transferTo (wifeAccount, 3000000);
}
};

// The wife wants to transfer 2 million from her account to her husband's account

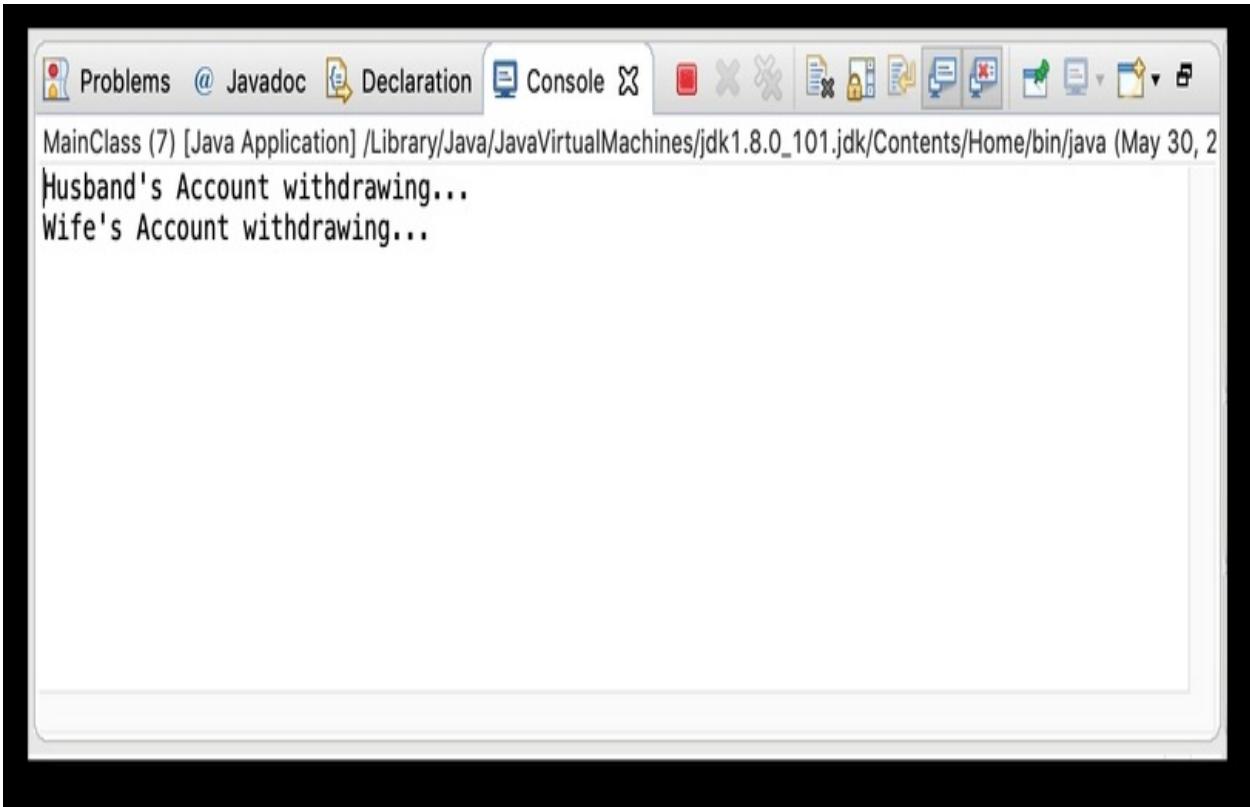
Thread wifeThread = new Thread () {
@Override
public void run () {

wifeAccount.transferTo (husbandAccount, 2000000);
}
};

// Two people make money transfer orders almost simultaneously
husbandThread.start ();
wifeThread.start ();

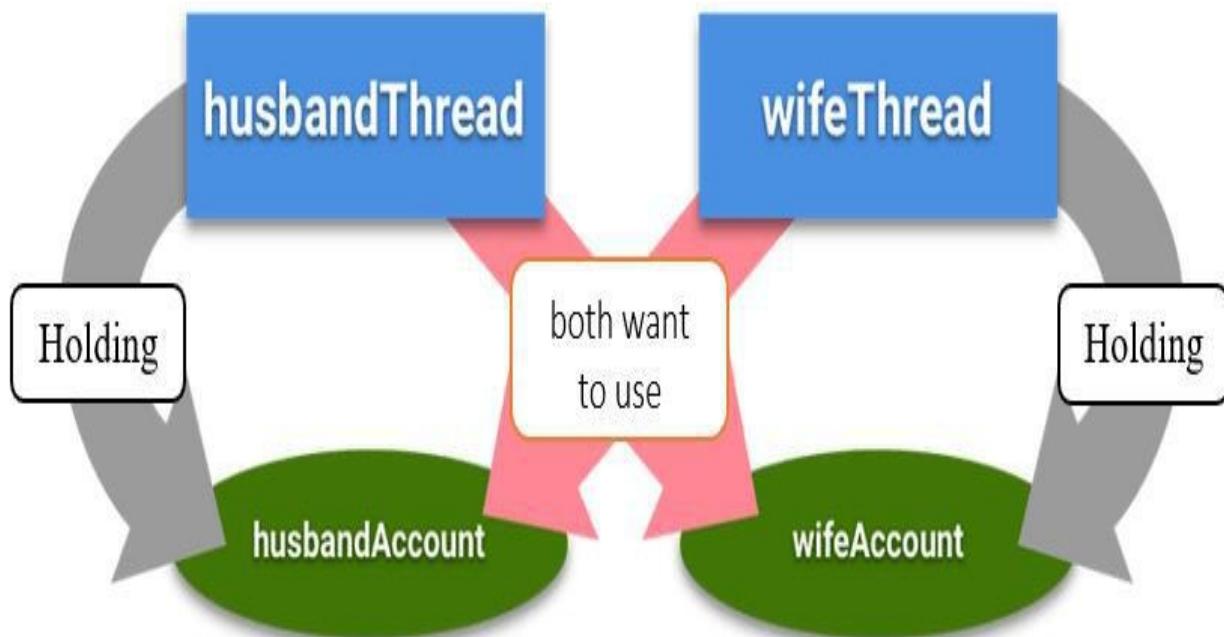
}
```

And here are the results from the program execution.



The results are printed to the console when the program is executed

As you also understand. You see, both Threads when launched, can only do the subtraction operation of the source account itself. And then to the top-up method for the target account ... can't be called. The application is still running at the moment, as evidenced by the red square *Stop* button next to the *Console* tab is still on, ie the application is still running and Eclipse is still allowing you to stop the application at any time. The application cannot be terminated forever because each Thread itself, when initialized, holds the *Lock on Monitor*. of an account, other Threads cannot interfere with the account each thread is holding. The fact that each Thread holds an account and waits for the turn to use another account (also being held by another Thread) is called *Deadlock* . It is similar to the following diagram.



Deadlock cause diagram of the above example

III. When Did Deadlock Appear?

As you were familiar with above, *Deadlock* often appears when we overdo the *synchronized* keyword . It causes Threads to hold the shared objects forever and not return them to other objects for use.

However, this lesson about *Deadlock* is just a theoretical lesson, in my opinion it is primarily a warning. In fact, it would be very difficult for a *Deadlock* to happen like this. However, while it is unlikely to happen, it has also happened, and it is our duty as developers that we still need to know and prepare the necessary knowledge about it.

IV. How To Avoid Deadlock And How To Handle It If Meeting Deadlock?

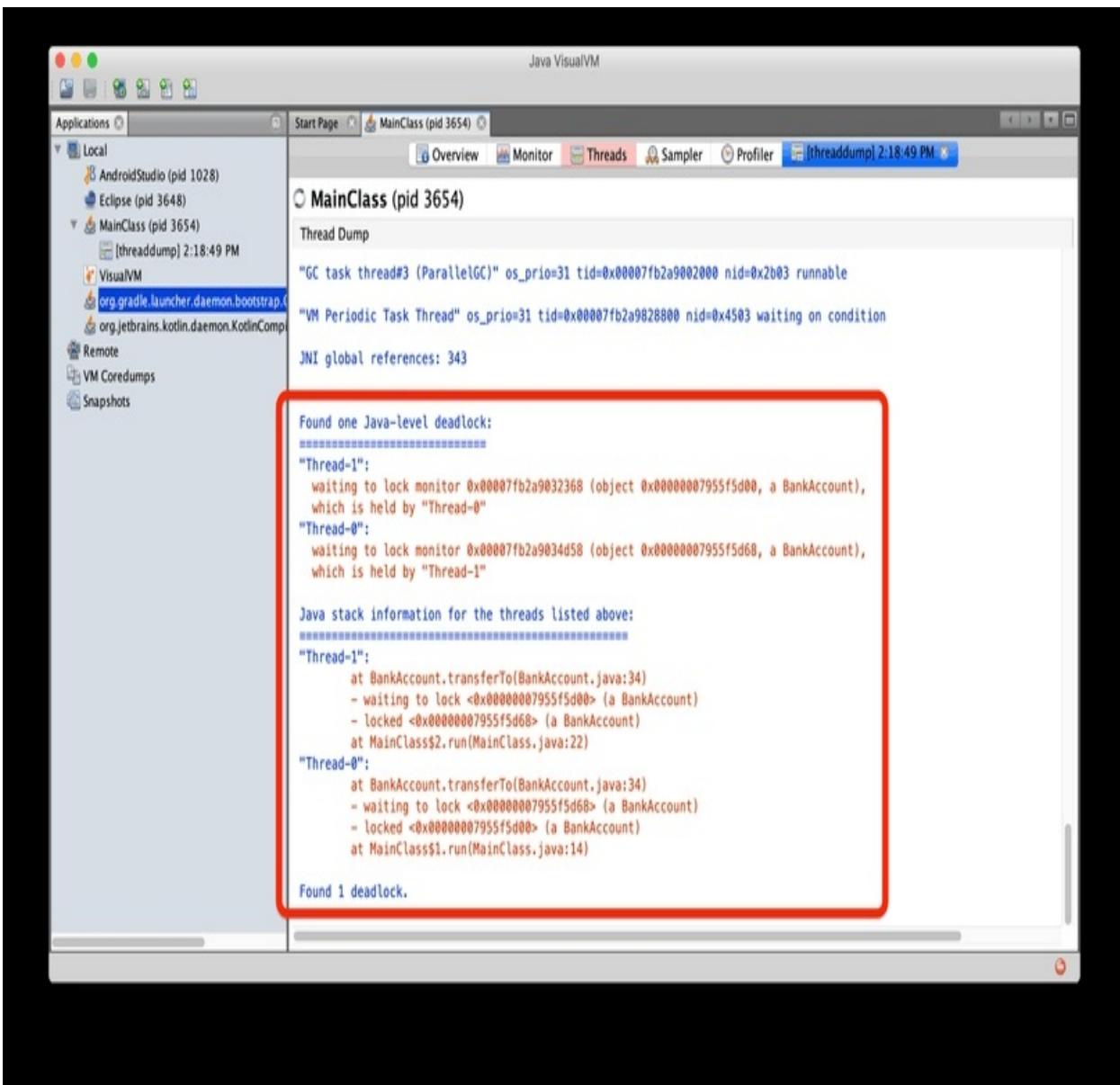
Like I said, *Deadlock* is very unlikely to happen in practice. Actually, in my programming life, I've never touched this case. As part of me know, our applications use many Threads but not enough and complex enough to cause

conflicts like the examples above.

But even if it is unlikely to happen. We still have to know to minimize the risk of this *deadlock* phenomenon . And even so avoid it, what if *Deadlock* one day happens to happen to your app? Then you should still have the knowledge ready to fix the source code and release the fix immediately. This section will cover how to avoid, and fix, errors for *Deadlock* .

First of all, in my *opinion* , to avoid *Deadlock* , you still need to understand your code. You must know what resources the used Threads use. Are there a lot of threads that are taking up shared resources? Ensuring the Threads are consuming resources will eventually return resources to the system as quickly as possible, so that other Threads have a chance to use and terminate the lives of those Threads. The easiest way is that you do not have to nest *synchronized* blocks as the above example is to ensure that *Deadlock* is very unlikely .

Then if your application executes in the real environment, that *deadlock* encounters . If your project is relatively small, you can also detect it by reading code and inference. But if the project is too large, you can use a number of tools with *Thread Dump* functionality . As shown below, I thanks to a tool available in the / bin directory of JDK, a tool called *jvisualvm* . How to use the tool and find the Thread Dump, you can refer to more on the internet, or you can see at *this link* to know all the tools, including *jvisualvm* .



Deadlock detection tool in threads and related resources

Deadlock lesson ends here. You can see that *Deadlock* is still related to the powerful Thread Knowledge Chain. Thereby you have seen the importance of Threads in Java, right?

This is the last lesson of Java Core. I hope you really confident in JAVA skills now. Good luck to all.