Ghi chú của một coder

Vũ Anh

Tháng 04 năm 2018

Mục lục

Mục lục **1**

1 Get Started **2**

2 Basic Syntax **3**
2.1 Variable Types 3

3 Data Structure **8**
3.0.1 Set 8

4 OOP **10**
4.0.1 Classes 10
4.0.2 Encapsulation 14
4.0.3 Inheritance 14
4.0.4 Polymorphism 16
4.0.5 Abstraction 18
4.1 File System IO 20
4.2 Error Handling 24
4.3 Logging 29
4.4 IDE 30
4.5 Package Manager 30
4.6 Build Tool 31
4.7 Production 31

Tài liệu **31**

01/11/2017: Java đơn giản là gay nhé. Không chơi. Viết java chỉ viết thế này thôi. Không viết hơn. Thề!
View online http://magizbox.com/training/java/site/
Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture. As of 2016, Java is one of the most popular programming languages in use, particularly for client-server

web applications, with a reported 9 million developers. Java was originally developed by James Gosling at Sun Microsystems (which has since been acquired by Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

Chương 1

Get Started

Installation Ubuntu Step 1. Download sdk
http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html
Step 2. Create folder jvm
sudo mkdir /usr/lib/jvm/ Step 3. cd to folder downloads jdk and run command
sudo mv jdk1.7.0$_x//usr/lib/jvm/jdk1.7.0_xRuninstalljavasudoupdate-alternatives-$
$-install/usr/bin/javajava/usr/lib/jvm/jdk1.7.0_x/jre/bin/java0Addpathjdk:$
$/usr/lib/jvm/jdk1.7.0_x$
su - nano /etc/environment

Chương 2

Basic Syntax

2.1 Variable Types

Although Java is object oriented, not all types are objects. It is built on top of basic variable types called primitives.
Here is a list of all primitives in Java:
byte (number, 1 byte) short (number, 2 bytes) int (number, 4 bytes) long (number, 8 bytes) float (float number, 4 bytes) double (float number, 8 bytes) char (a character, 2 bytes) boolean (true or false, 1 byte) Java is a strong typed language, which means variables need to be defined before we use them. Numbers To declare and assign a number use the following syntax:
int myNumber; myNumber = 5; Or you can combine them:
int myNumber = 5; To define a double floating point number, use the following syntax:
double d = 4.5; d = 3.0; If you want to use float, you will have to cast:
float f = (float) 4.5; Or, You can use this:
float f = 4.5f (f is a shorter way of casting float) Characters and Strings In Java, a character is it's own type and it's not simply a number, so it's not common to put an ascii value in it, there is a special syntax for chars:
char c = 'g'; String is not a primitive. It's a real type, but Java has special treatment for String.
Here are some ways to use a string:
// Create a string with a constructor String s1 = new String("Who let the dogs out?"); // Just using "" creates a string, so no need to write it the previous way. String s2 = "Who who who who!"; // Java defined the operator + on strings to concatenate: String s3 = s1 + s2; There is no operator overloading in Java! The operator + is only defined for strings, you will never see it with other objects, only primitives.
You can also concat string to primitives:
int num = 5; String s = "I have " + num + " cookies"; //Be sure not to use "" with primitives. boolean Every comparison operator in java will return the type boolean that not like other languages can only accept two special values: true or false.
boolean b = false; b = true;
boolean toBe = false; b = toBe || !toBe; if (b) System.out.println(toBe);

int children = 0; b = children; // Will not work if (children) // Will not work // Will not work Operators Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:
Arithmetic Operators Relational Operators Bitwise Operators Logical Operators Assignment Operators Misc Operators The Arithmetic Operators Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.
The following table lists the arithmetic operators:
Operator Description Example + (Addition) Adds values on either side of the operator 10 + 20 -> 30 - (Subtraction) Subtracts right hand operand from left hand operand 10 - 20 -> -10 * (Multiplication) Multiplies values on either side of the operator 10 * 20 -> 200 / (Division) Divides left hand operand by right hand operand 20 / 10 -> 2 ++ (Increment) Increases the value of operand by 1 a = 20
a++ -> 21
– (Decrement) Decreases the value of operand by 1 a = 20
a– -> 19
The Relational Operators There are following relational operators supported by Java language
== (equal to) Checks if the values of two operands are equal or not, if yes then condition becomes true.
Example: (A == B) is not true. 2 != (not equal to) Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
Example: (A != B) is true.
3 > (greater than) Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
Example: (A > B) is not true. 4 < (less than) Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
Example: (A < B) is true. 5 >= (greater than or equal to) Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
Example (A >= B) is not true. 6 <= (less than or equal to) Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.
example(A <= B) is true.
The Bitwise Operators Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.
Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:
a = 0011 1100
b = 0000 1101
ab = 0000 1100
a|b = 0011 1101
a^b = 00110001
 a = 1100 0011
The following table lists the bitwise operators:
Assume integer variable A holds 60 and variable B holds 13 then:
(bitwise and) Binary AND Operator copies a bit to the result if it exists in both operands.

Example: (A B) will give 12 which is 0000 1100 2 | (bitwise or) Binary OR Operator copies a bit if it exists in either operand.
Example: (A | B) will give 61 which is 0011 1101 3 $^{(}bitwiseXOR)BinaryXOROperatorcopiesthebitifitissetin$
Example: (A $^{B})willgive49whichis001100014\ (bitwisecompliment)BinaryOnesComplementOperatorisunar$
Example: (A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. 5 « (left shift) Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand
Example: A « 2 will give 240 which is 1111 0000 6 » (right shift) Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.
Example: A » 2 will give 15 which is 1111 7 »> (zero fill right shift) Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.
Example: A »>2 will give 15 which is 0000 1111
The Logical Operators The following table lists the logical operators:
Assume Boolean variables A holds true and variable B holds false, then:
(logical and) Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.
Example (A B) is false. 2 || (logical or) Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.
Example (A || B) is true. 3 ! (logical not) Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.
Example !(A B) is true.
The Assignment Operators There are following assignment operators supported by Java language:
Show Examples
SR.NO Operator and Description 1 = Simple assignment operator, Assigns values from right side operands to left side operand.
Example: C = A + B will assign value of A + B into C 2 += Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.
Example: C += A is equivalent to C = C + A 3 -= Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.
Example:C -= A is equivalent to C = C - A 4 *= Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.
Example: C *= A is equivalent to C = C * A 5 /= Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand
ExampleC /= A is equivalent to C = C / A 6
Example: C
ExampleC «= 2 is same as C = C « 2 8 »= Right shift AND assignment operator
Example C »= 2 is same as C = C » 2 9 = Bitwise AND assignment operator.
Example: C = 2 is same as C = C 2 10 $^{=}bitwiseexclusiveORandassignmentoperator.$
Example: C $^{=}2issameasC = C^{2}11| = bitwiseinclusiveORandassignmentoperator.$
Example: C |= 2 is same as C = C | 2
Miscellaneous Operators There are few other operators supported by Java Language.

Conditional Operator (? :) Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:
variable x = (expression) ? value if true : value if false Following is the example:
public class Test
public static void main(String args[]) int a, b; a = 10; b = (a == 1) ? 20: 30; System.out.println("Value of b is : " + b);
b = (a == 10) ? 20: 30; System.out.println("Value of b is : " + b); This would produce the following result ?
Value of b is : 30 Value of b is : 20 Precedence of Operators Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:
For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.
Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.
Category Operator Associativity Postfix () [] . (dot operator) Left toright Unary ++ - - ! Right to left Multiplicative * /
Conditional Java uses boolean variables to evaluate conditions. The boolean values true and false are returned when an expression is compared or evaluated. For example:
int a = 4; boolean b = a == 4;
if (b) System.out.println("It's true!"); Of course we don't normally assign a conditional expression to a boolean, we just use the short version:
int a = 4;
if (a == 4) System.out.println("Ohhh! So a is 4!"); Boolean operators There aren't that many operators to use in conditional statements and most of them are pretty strait forward:
int a = 4; int b = 5; boolean result; result = a < b; // true result = a > b; // false result = a <= 4 // a smaller or equal to 4 - true result = b >= 6 // b bigger or equal to 6 - false result = a == b // a equal to b - false result = a != b // a is not equal to b - true result = a > b || a < b // Logical or - true result = 3 < a a < 6 // Logical and - true result = !result // Logical not - false if - else and between The if, else statement in java is pretty simple.
if (a == b) // a and b are equal, let's do something cool And we can also add an else statement after an if, to do something if the condition is not true
if (a == b) // We already know this part else // a and b are not equal... :/
The if - else statements doesn't have to be in several lines with , if can be used in one line, or without the , for a single line statment.
if (a == b) System.out.println("Another line Wow!"); else System.out.println("Double rainbow!"); Although this method might be useful for making your code shorter by using fewer lines, we strongly recommend for beginners not to use this short version of statements and always use the full version with . This goes to every statement that can be shorted to a single line (for, while, etc).

The ugly side of if There is a another way to write a one line if - else statement by using the operator ? :
int a = 4; int result = a == 4 ? 1 : 8;
// result will be 1 // This is equivalent to int result;
if (a == 4) result = 1; else result = 8; Again, we strongly recommend for beginners not to use this version of if.
== and equals The operator == works a bit different on objects than on primitives. When we are using objects and want to check if they are equal, the operator == will say if they are the same, if you want to check if they are logically equal, you should use the equals method on the object. For example:
String a = new String("Wow"); String b = new String("Wow"); String sameA = a;
boolean r1 = a == b; // This is false, since a and b are not the same object boolean r2 = a.equals(b); // This is true, since a and b are logically equals boolean r3 = a == sameA; // This is true, since a and sameA are really the same object

Chương 3

Data Structure

Data Structure Number, String Convert number to string
String.valueOf(1000) Make a random
// create a random number from 0 to 99 (new Random()).nextInt(100) Collection Arrays Arrays in Java are also objects. They need to be declared and then created. In order to declare a variable that will hold an array of integers, we use the following syntax:
int[] arr; Notice there is no size, since we didn't create the array yet.
arr = new int[10]; This will create a new array with the size of 10. We can check the size by printing the array's length:
System.out.println(arr.length); We can access the array and set values:
arr[0] = 4; arr[1] = arr[0] + 5; Java arrays are 0 based, which means the first element in an array is accessed at index 0 (e.g: arr[0], which accesses the first element). Also, as an example, an array of size 5 will only go up to index 4 due to it being 0 based.
int[] arr = new int[5] //accesses and sets the first element arr[0] = 4; We can also create an array with values in the same line:
int[] arr = 1, 2, 3, 4, 5; Don't try to print the array without a loop, it will print something nasty like [I@f7e6a96.

3.0.1 Set

import java.util.HashSet; import java.util.Set;
public class HelloWorld
public static void main(String []args) Set<Dog> dogs = new HashSet<Dog>(); Dog dog1 = new Dog("a", 1); Dog dog2 = new Dog("a", 2); Dog dog3 = new Dog("a", 1); Dog dog4 = new Dog("b", 1); dogs.add(dog1); dogs.add(dog2); dogs.add(dog3); dogs.add(dog4); System.out.println(dogs.size());
// 3 public class Dog public String name; public int age; public int value; public Dog(String name, int age) this.name = name; this.age = age; value = (this.name + String.valueOf(this.age)).hashCode();
@Override public int hashCode() return value;
@Override public boolean equals(Object obj) return (obj instanceof Dog ((Dog) obj).value == this.value); List List<String> places = Arrays.asList("Buenos Aires", "Córdoba", "La Plata"); Datetime Calendar c = Calendar.getInstance();

Suggest Readings Initialization of an ArrayList in one line How to convert from int to String?

Chương 4

OOP

4.0.1 Classes

Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts

1. Classes and Objects
2. Inheritance
3. Polymorphism
4. Abstraction
5. Instance
6. Method
7. Message Parsing

In this chapter, we will look into the concepts - Classes and Objects.
Object Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class. Class A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support. Objects Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.
If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.
If you compare the software object with a real-world object, they have very similar characteristics.
Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.
So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.
Classes A class is a blueprint from which individual objects are created.
Following is a sample of a class.
Example

public class Dog String breed; int ageC String color;
void barking()
void hungry()
void sleeping()   A class can contain any of the following variable types.
Local variables Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed. Instance variables Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class. Class variables Class variables are variables declared within a class, outside any method, with the static keyword. A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.
Following are some of the important topics that need to be discussed when looking into classes of the Java Language.
Constructors When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.
Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.
Following is an example of a constructor
Example
public class Puppy public Puppy()
public Puppy(String name) // This constructor has one parameter, name. Java also supports Singleton Classes where you would be able to create only one instance of a class.
Note We have two different types of constructors. We are going to discuss constructors in detail in the subsequent chapters.
Creating an Object As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.
There are three steps when creating an object from a class
Declaration A variable declaration with a variable name with an object type. Instantiation The 'new' keyword is used to create the object. Initialization The 'new' keyword is followed by a call to a constructor. This call initializes the new object. Following is an example of creating an object
Example

```java
public class Puppy {
   public Puppy(String name) {
      // This constructor has one parameter, name.
      System.out.println("Passed Name is :" + name );
   }

   public static void main(String []args) {
      // Following statement would create an object myPuppy
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

If we compile and run the above program, then it will produce the following result
Passed Name is :tommy Accessing Instance Variables and Methods Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path
/* First create an object */ ObjectReference = new Constructor();
/* Now call a variable as follows */ ObjectReference.variableName;
/* Now you can call a class method as follows */ ObjectReference.MethodName();
Example
This example explains how to access instance variables and methods of a class.
public class Puppy int puppyAge;
public Puppy(String name) // This constructor has one parameter, name. System.out.println("Name chosen is :" + name);
public void setAge(int age) puppyAge = age;
public int getAge( ) System.out.println("Puppy's age is :" + puppyAge); return puppyAge;
public static void main(String []args) /* Object creation */ Puppy myPuppy = new Puppy("tommy");
/* Call class method to set puppy's age */ myPuppy.setAge(2);
/* Call another class method to get puppy's age */ myPuppy.getAge( );
/* You can access instance variable as follows as well */ System.out.println("Variable Value :" + myPuppy.puppyAge);   If we compile and run the above program, then it will produce the following result
Output
Name chosen is :tommy Puppy's age is :2 Variable Value :2 Source File Declaration Rules As the last part of this section, let's now look into the source file declaration rules. These rules are essential when declaring classes, import statements and package statements in a source file.
There can be only one public class per source file. A source file can have multiple non-public classes. The public class name should be the name of the source file as well which should be appended by .java at the end. For example: the class name is public class Employee then the source file should be as Employee.java. If the class is defined inside a package, then the package statement should be the first statement in the source file. If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file. Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file. Classes have several access levels and there are different types of classes; abstract classes, final classes, etc. We will be explaining about all these in the access modifiers chapter.
Apart from the above mentioned types of classes, Java also has some special classes called Inner classes and Anonymous classes.
Java Package In simple words, it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces

will be written, therefore categorizing these classes is a must as well as makes life much easier.
Import Statements In Java if a fully qualified name, which includes the package and the class name is given, then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.
For example, the following line would ask the compiler to load all the classes available in directory java$_installation/java/io$
import java.io.*; A Simple Case Study For our case study, we will be creating two classes. They are Employee and EmployeeTest.
First open notepad and add the following code. Remember this is the Employee class and the class is a public class. Now, save this source file with the name Employee.java.
The Employee class has four instance variables - name, age, designation and salary. The class has one explicitly defined constructor, which takes a parameter.
Example
import java.io.*; public class Employee
String name; int age; String designation; double salary;
// This is the constructor of the class Employee public Employee(String name)
this.name = name;
// Assign the age of the Employee to the variable age. public void empAge(int empAge) age = empAge;
/* Assign the designation to the variable designation.*/ public void empDesignation(String empDesig) designation = empDesig;
/* Assign the salary to the variable salary.*/ public void empSalary(double empSalary) salary = empSalary;
/* Print the Employee details */ public void printEmployee() System.out.println("Name:"+ name); System.out.println("Age:" + age); System.out.println("Designation:" + designation); System.out.println("Salary:" + salary);   As mentioned previously in this tutorial, processing starts from the main method. Therefore, in order for us to run this Employee class there should be a main method and objects should be created. We will be creating a separate class for these tasks.
Following is the EmployeeTest class, which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.
Save the following code in EmployeeTest.java file.
import java.io.*; public class EmployeeTest
public static void main(String args[]) /* Create two objects using constructor */ Employee empOne = new Employee("James Smith"); Employee empTwo = new Employee("Mary Anne");
// Invoking methods for each object created empOne.empAge(26); empOne.empDesignation("Senior Software Engineer"); empOne.empSalary(1000); empOne.printEmployee();
empTwo.empAge(21); empTwo.empDesignation("Software Engineer"); empTwo.empSalary(500);
empTwo.printEmployee();   Now, compile both the classes and then run EmployeeTest to see the result as follows
Output
C:$javacEmployee.javaC : javacEmployeeTest.javaC : javaEmployeeTestName : JamesSmithAge : 26Designation : SeniorSoftwareEngineerSalary : 1000.0Name : MaryAnneAge : 21Designation : SoftwareEngineerSalary : 500.0$

4.0.2 Encapsulation

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.
Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.
Implementation To achieve encapsulation in Java
Declare the variables of a class as private. Provide public setter and getter methods to modify and view the variables values. Example Following is an example that demonstrates how to achieve Encapsulation in Java
/* File name : EncapTest.java */ public class EncapTest private String name; private String idNum; private int age;
public int getAge() return age;
public String getName() return name;
public String getIdNum() return idNum;
public void setAge(int newAge) age = newAge;
public void setName(String newName) name = newName;
public void setIdNum(String newId) idNum = newId; The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.
The variables of the EncapTest class can be accessed using the following program
/* File name : RunEncap.java */ public class RunEncap
public static void main(String args[]) EncapTest encap = new EncapTest(); encap.setName("James"); encap.setAge(20); encap.setIdNum("12343ms"); System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge());
This will produce the following result
Name : James Age : 20 Benefits The fields of a class can be made read-only or write-only. A class can have total control over what is stored in its fields. The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code. Related Readings "Java Inheritance". www.tutorialspoint.com. N.p., 2016. Web. 10 Dec. 2016.

4.0.3 Inheritance

In the preceding lessons, you have seen inheritance mentioned several times. In the Java language, classes can be derived from other classes, thereby inheriting fields and methods from those classes.
The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
Class Hierarchy The Object class, defined in the java.lang package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from Object, other classes derive from some of those classes, and so on, forming a hierarchy of classes.
At the top of the hierarchy, Object is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.
An Example Here is the sample code for a possible implementation of a Bicycle class that was presented in the Classes and Objects lesson:
public class Bicycle
// the Bicycle class has three fields public int cadence; public int gear; public int speed;
// the Bicycle class has one constructor public Bicycle(int startCadence, int startSpeed, int startGear) gear = startGear; cadence = startCadence; speed = startSpeed;
// the Bicycle class has four methods public void setCadence(int newValue) cadence = newValue;
public void setGear(int newValue) gear = newValue;
public void applyBrake(int decrement) speed -= decrement;
public void speedUp(int increment) speed += increment;
A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:
public class MountainBike extends Bicycle
// the MountainBike subclass adds one field public int seatHeight;
// the MountainBike subclass has one constructor public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) super(startCadence, startSpeed, startGear); seatHeight = startHeight;
// the MountainBike subclass adds one method public void setHeight(int newValue) seatHeight = newValue; MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it. Except for the constructor, it is as if you had written a new MountainBike class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the methods in the Bicycle class were complex and had taken substantial time to debug.
What You Can Do in a Subclass A subclass inherits all of the public and protected members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the package-private members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:
The inherited fields can be used directly, just like any other fields. You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not * recommended). You can declare new fields in the subclass that are not in the superclass. The inherited methods can be used directly as they are. You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it. You can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it. You can declare new methods in the subclass that are not in the superclass. You can write a subclass constructor that invokes the

constructor of the superclass, either implicitly or by using the keyword super. The following sections in this lesson will expand on these topics.
Private Members in a Superclass A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.
A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.
Casting Objects We have seen that an object is of the data type of the class from which it was instantiated. For example, if we write
public MountainBike myBike = new MountainBike(); then myBike is of type MountainBike.
MountainBike is descended from Bicycle and Object. Therefore, a MountainBike is a Bicycle and is also an Object, and it can be used wherever Bicycle or Object objects are called for.
The reverse is not necessarily true: a Bicycle may be a MountainBike, but it isn't necessarily. Similarly, an Object may be a Bicycle or a MountainBike, but it isn't necessarily.
Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write
Object obj = new MountainBike(); then obj is both an Object and a MountainBike (until such time as obj is assigned another object that is not a MountainBike). This is called implicit casting.
If, on the other hand, we write
MountainBike myBike = obj; we would get a compile-time error because obj is not known to the compiler to be a MountainBike. However, we can tell the compiler that we promise to assign a MountainBike to obj by explicit casting:
MountainBike myBike = (MountainBike)obj; This cast inserts a runtime check that obj is assigned a MountainBike so that the compiler can safely assume that obj is a MountainBike. If obj is not a MountainBike at runtime, an exception will be thrown.
Related Readings "Inheritance". docs.oracle.com. N.p., 2016. Web. 8 Dec. 2016. "Java Inheritance". www.tutorialspoint.com. N.p., 2016. Web. 8 Dec. 2016. Friesen, Jeff. "Java 101: Inheritance In Java, Part 1". JavaWorld. N.p., 2016. Web. 8 Dec. 2016.

4.0.4 Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.
It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.
The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods

that it can invoke on the object.
A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.
Example Let us look at an example.
public interface Vegetarian public class Animal public class Deer extends Animal implements Vegetarian Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples
A Deer IS-A Animal A Deer IS-A Vegetarian A Deer IS-A Deer A Deer IS-A Object When we apply the reference variable facts to a Deer object reference, the following declarations are legal
Deer d = new Deer(); Animal a = d; Vegetarian v = d; Object o = d; All the reference variables d, a, v, o refer to the same Deer object in the heap.
Virtual Methods In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.
We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.
/* File name : Employee.java */ public class Employee private String name; private String address; private int number;
public Employee(String name, String address, int number) System.out.println("Constructing an Employee"); this.name = name; this.address = address; this.number = number;
public void mailCheck() System.out.println("Mailing a check to " + this.name + " " + this.address);
public String toString() return name + " " + address + " " + number;
public String getName() return name;
public String getAddress() return address;
public void setAddress(String newAddress) address = newAddress;
public int getNumber() return number; Now suppose we extend Employee class as follows
/* File name : Salary.java */ public class Salary extends Employee private double salary; // Annual salary
public Salary(String name, String address, int number, double salary) super(name, address, number); setSalary(salary);
public void mailCheck() System.out.println("Within mailCheck of Salary class "); System.out.println("Mailing check to " + getName() + " with salary " + salary);
public double getSalary() return salary;
public void setSalary(double newSalary) if(newSalary >= 0.0) salary = newSalary;

public double computePay() System.out.println("Computing salary pay for " + getName()); return salary/52; Now, you study the following program carefully and try to determine its output
/* File name : VirtualDemo.java */ public class VirtualDemo
public static void main(String [] args) Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00); Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00); System.out.println("Call mailCheck using Salary

reference –"); s.mailCheck(); System.out.println("Call mailCheck using Employee reference–"); e.mailCheck(); This will produce the following result
Constructing an Employee Constructing an Employee
Call mailCheck using Salary reference – Within mailCheck of Salary class Mailing check to Mohd Mohtashim with salary 3600.0
Call mailCheck using Employee reference– Within mailCheck of Salary class Mailing check to John Adams with salary 2400.0 Here, we instantiate two Salary objects. One using a Salary reference s, and the other using an Employee reference e.
While invoking s.mailCheck(), the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.
mailCheck() on e is quite different because e is an Employee reference. When the compiler sees e.mailCheck(), the compiler sees the mailCheck() method in the Employee class.
Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.
This behavior is referred to as virtual method invocation, and these methods are referred to as virtual methods. An overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.
Related Readings "Java Polymorphism". www.tutorialspoint.com. N.p., 2016. Web. 10 Dec. 2016.

4.0.5 Abstraction

As per dictionary, abstraction is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.
Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.
In Java, abstraction is achieved using Abstract classes and interfaces.
Abstract Class A class which contains the abstract keyword in its declaration is known as abstract class.
Abstract classes may or may not contain abstract methods, i.e., methods without body (public void get();) But, if a class has at least one abstract method, then the class must be declared abstract. If a class is declared abstract, it cannot be instantiated. To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it. If you inherit an abstract class, you have to provide implementations to all the abstract methods in it. Example
This section provides you an example of the abstract class. To create an abstract class, just use the abstract keyword before the class keyword, in the class declaration.

/* File name : Employee.java */ public abstract class Employee private String name; private String address; private int number;
public Employee(String name, String address, int number) System.out.println("Constructing an Employee"); this.name = name; this.address = address; this.number = number;
public double computePay() System.out.println("Inside Employee computePay"); return 0.0;
public void mailCheck() System.out.println("Mailing a check to " + this.name + " " + this.address);
public String toString() return name + " " + address + " " + number;
public String getName() return name;
public String getAddress() return address;
public void setAddress(String newAddress) address = newAddress;
public int getNumber() return number; You can observe that except abstract methods the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor.
Now you can try to instantiate the Employee class in the following way
/* File name : AbstractDemo.java */ public class AbstractDemo
public static void main(String [] args) /* Following is not allowed and would raise error */ Employee e = new Employee("George W.", "Houston, TX", 43);
System.out.println("Call mailCheck using Employee reference–"); e.mailCheck();
When you compile the above class, it gives you the following error
Employee.java:46: Employee is abstract; cannot be instantiated Employee e = new Employee("George W.", "Houston, TX", 43); $^{1}errorInheritingtheAbstractClassWecaninherittheprope$
/* File name : Salary.java */ public class Salary extends Employee private double salary; // Annual salary
public Salary(String name, String address, int number, double salary) super(name, address, number); setSalary(salary);
public void mailCheck() System.out.println("Within mailCheck of Salary class "); System.out.println("Mailing check to " + getName() + " with salary " + salary);
public double getSalary() return salary;
public void setSalary(double newSalary) if(newSalary >= 0.0) salary = newSalary;

public double computePay() System.out.println("Computing salary pay for " + getName()); return salary/52; Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below.
/* File name : AbstractDemo.java */ public class AbstractDemo
public static void main(String [] args) Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00); Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00); System.out.println("Call mailCheck using Salary reference –"); s.mailCheck(); System.out.println("mailCheck using Employee reference–"); e.mailCheck(); This produces the following result
Constructing an Employee Constructing an Employee Call mailCheck using Salary reference – Within mailCheck of Salary class Mailing check to Mohd Mohtashim with salary 3600.0
Call mailCheck using Employee reference– Within mailCheck of Salary class Mailing check to John Adams with salary 2400.0 Abstract Methods If you want a class to contain a particular method but you want the actual implementation

of that method to be determined by child classes, you can declare the method in the parent class as an abstract.
abstract keyword is used to declare the method as abstract. You have to place the abstract keyword before the method name in the method declaration. An abstract method contains a method signature, but no method body. Instead of curly braces, an abstract method will have a semoi colon (;) at the end. Following is an example of the abstract method.
public abstract class Employee private String name; private String address; private int number;
public abstract double computePay(); // Remainder of class definition Declaring a method as abstract has two consequences
The class containing it must be declared as abstract. Any class inheriting the current class must either override the abstract method or declare itself as abstract. Note Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.
Suppose Salary class inherits the Employee class, then it should implement the computePay() method as shown below
/* File name : Salary.java */ public class Salary extends Employee private double salary; // Annual salary
public double computePay() System.out.println("Computing salary pay for " + getName()); return salary/52; // Remainder of class definition Related Readings "Java Abstraction". www.tutorialspoint.com. N.p., 2016. Web. 10 Dec. 2016.

4.1 File System IO

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.
Stream A stream can be defined as a sequence of data. There are two kinds of Streams
InPutStream The InputStream is used to read data from a source. OutPutStream The OutputStream is used for writing data to a destination.
Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one
Byte Streams Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream. Following is an example which makes use of these two classes to copy an input file into an output file
Example
import java.io.*; public class CopyFile
public static void main(String args[]) throws IOException FileInputStream in = null; FileOutputStream out = null;
try in = new FileInputStream("input.txt"); out = new FileOutputStream("output.txt");

int c; while ((c = in.read()) != -1) out.write(c); finally if (in != null) in.close(); if (out != null) out.close(); Now let's have a file input.txt with the following content
This is test for copy file. As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following $javacCopyFile.java$java CopyFile Character Streams Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time. We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file
Example
import java.io.*; public class CopyFile
public static void main(String args[]) throws IOException FileReader in = null; FileWriter out = null;
try in = new FileReader("input.txt"); out = new FileWriter("output.txt");
int c; while ((c = in.read()) != -1) out.write(c); finally if (in != null) in.close(); if (out != null) out.close(); Now let's have a file input.txt with the following content
This is test for copy file. As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following $javacCopyFile.java$java CopyFile Standard Streams All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams
Standard Input This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in. Standard Output This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as System.out. Standard Error This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err. Following is a simple program, which creates InputStreamReader to read standard input stream until the user types a "q"
Example
import java.io.*; public class ReadConsole
public static void main(String args[]) throws IOException InputStreamReader cin = null;
try cin = new InputStreamReader(System.in); System.out.println("Enter characters, 'q' to quit."); char c; do c = (char) cin.read(); System.out.print(c); while(c != 'q'); finally if (cin != null) cin.close(); Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in

the following program. This program continues to read and output the same character until we press 'q'
*javacReadConsole.java*java ReadConsole Enter characters, 'q' to quit. 1 1 e e q q Reading and Writing Files As described earlier, a stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.
Here is a hierarchy of classes to deal with Input and Output streams.
The two important streams are FileInputStream and FileOutputStream, which would be discussed in this tutorial.
FileInputStream This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.
Following constructor takes a file name as a string to create an input stream object to read the file
InputStream f = new FileInputStream("C:/java/hello"); Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows
File f = new File("C:/java/hello"); InputStream f = new FileInputStream(f);
Once you have InputStream object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.
Method Description 1 public void close() throws IOException
This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2 protected void finalize()throws IOException
This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3 public int read(int r)throws IOException
This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4 public int read(byte[] r) throws IOException
This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
5 public int available() throws IOException
Gives the number of bytes that can be read from this file input stream. Returns an int.
There are other important input streams available, for more detail you can refer to the following links
ByteArrayInputStream DataInputStream FileOutputStream FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.
Here are two constructors which can be used to create a FileOutputStream object.
Following constructor takes a file name as a string to create an input stream object to write the file
OutputStream f = new FileOutputStream("C:/java/hello") Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows

File f = new File("C:/java/hello"); OutputStream f = new FileOutputStream(f);
Once you have OutputStream object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.
Method Description 1 public void close() throws IOException
This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2 protected void finalize()throws IOException
This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3 public void write(int w)throws IOException
This methods writes the specified byte to the output stream.
4 public void write(byte[] w)
Writes w.length bytes from the mentioned byte array to the OutputStream.
There are other important output streams available, for more detail you can refer to the following links
ByteArrayOutputStream DataOutputStream Example
Following is the example to demonstrate InputStream and OutputStream
import java.io.*; public class fileStreamTest
public static void main(String args[])
try byte bWrite [] = 11,21,3,40,5; OutputStream os = new FileOutputStream("test.txt");
for(int x = 0; x < bWrite.length ; x++) os.write(bWrite[x]); // writes the bytes os.close();
InputStream is = new FileInputStream("test.txt"); int size = is.available();
for(int i = 0; i < size; i++) System.out.print((char)is.read() + " "); is.close();
catch(IOException e) System.out.print("Exception"); The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.
File Navigation and I/O There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.
File Class FileReader Class FileWriter Class Directories in Java A directory is a File which can contain a list of other files and directories. You use File object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on File object and what are related to directories.
Creating Directories There are two useful File utility methods, which can be used to create directories
The mkdir( ) method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
The mkdirs() method creates both a directory and all the parents of the directory.
Following example creates "/tmp/user/java/bin" directory
Example
import java.io.File; public class CreateDir
public static void main(String args[]) String dirname = "/tmp/user/java/bin";
File d = new File(dirname);

// Create directory now. d.mkdirs(); Compile and execute the above code to create "/tmp/user/java/bin".
Note Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.
Listing Directories You can use list( ) method provided by File object to list down all the files and directories available in a directory as follows
Example
import java.io.File; public class ReadDir
public static void main(String[] args) File file = null; String[] paths;
try // create new file object file = new File("/tmp");
// array of files and directory paths = file.list();
// for each name in the path array for(String path:paths) // prints filename and directory name System.out.println(path); catch(Exception e) // if any error occurs e.printStackTrace(); This will produce the following result based on the directories and files available in your /tmp directory
test1.txt test2.txt ReadDir.java ReadDir.class Related Readings "Java Files And I/O". www.tutorialspoint.com. N.p., 2016. Web. 15 Dec. 2016.

4.2 Error Handling

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
An exception can occur for many different reasons. Following are some scenarios where an exception occurs.
A user has entered an invalid data. A file that needs to be opened cannot be found. A network connection has been lost in the middle of communications or the JVM has run out of memory. Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.
Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.
Type of exceptions Checked Exception
A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.
For example, if you use FileReader class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a FileNotFoundException occurs, and the compiler prompts the programmer to handle the exception.

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

If you try to compile the above program, you will get the following exceptions.
C:$javacFilenotFound_Demo.javaFilenotFound_Demo.java : 8 : error : unreportedexceptionFileNotFound$
$newFileReader(file);^1 errorNoteSincethemethodsread()andclose()ofFileReaderclassthrowsIOException$
Unchecked exceptions
An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an ArrayIndexOutOfBoundsExceptionexception occurs.
public class Unchecked$_Demo$
public static void main(String args[]) int num[] = 1, 2, 3, 4; System.out.println(num[5]);
If you compile and execute the above program, you will get the following exception.
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5 at Exceptions.Unchecked$_Demo.main(Unchecked_Demo.java : 8)Errors$
These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.
Exception Hierarchy All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.
Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.
The Exception class has two main subclasses: IOException class and RuntimeException Class.
Following is a list of most common checked and unchecked Java's Built-in Exceptions
Exceptions Methods Following is the list of important methods available in the Throwable class.
1 public String getMessage() Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor. 2 public Throwable getCause() Returns the cause of the exception as represented by a Throwable object. 3 public String toString() Returns the name of the class concatenated with the result of getMessage(). 4 public void printStackTrace() Prints the result of toString() along with the stack trace to System.err, the error output stream. 5 public StackTraceElement [] getStackTrace() Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack. 6 public Throwable fillInStackTrace() Fills the stack trace of this Throwable object with the current stack trace, adding to any

previous information in the stack trace. Catching Exceptions A method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following
Syntax
try // Protected code catch(ExceptionName e1) // Catch block The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.
A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.
Example
The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.
// File Name : ExcepTest.java import java.io.*;
public class ExcepTest
public static void main(String args[]) try int a[] = new int[2]; System.out.println("Access element three :" + a[3]); catch(ArrayIndexOutOfBoundsException e) System.out.println("Exception thrown :" + e); System.out.println("Out of the block"); This will produce the following result
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3 Out of the block Multiple Catch Blocks A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following
try // Protected code catch(ExceptionType1 e1) // Catch block catch(ExceptionType2 e2) // Catch block catch(ExceptionType3 e3) // Catch block The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.
Example
Here is code segment showing how to use multiple try/catch statements.
try file = new FileInputStream(fileName); x = (byte) file.read(); catch(IOException i) i.printStackTrace(); return -1; catch(FileNotFoundException f) // Not valid! f.printStackTrace(); return -1; Catching Multiple Type of Exceptions Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code. Here is how you would do it
catch (IOException|FileNotFoundException ex) logger.log(ex); throw ex; The Throws/Throw Keywords If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the throw keyword.
Try to understand the difference between throws and throw keywords, throws is used to postpone the handling of a checked exception and throw is used to invoke an exception explicitly.
The following method declares that it throws a RemoteException
import java.io.*; public class className
public void deposit(double amount) throws RemoteException // Method implementation throw new RemoteException(); // Remainder of class definition
A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException
import java.io.*; public class className
public void withdraw(double amount) throws RemoteException, InsufficientFundsException // Method implementation // Remainder of class definition
The Finally Block The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.
Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.
A finally block appears at the end of the catch blocks and has the following syntax
Syntax
try // Protected code catch(ExceptionType1 e1) // Catch block catch(ExceptionType2 e2) // Catch block catch(ExceptionType3 e3) // Catch block finally // The finally block always executes.
Example
public class ExcepTest
public static void main(String args[]) int a[] = new int[2]; try System.out.println("Access element three :" + a[3]); catch(ArrayIndexOutOfBoundsException e) System.out.println("Exception thrown :" + e); finally a[0] = 6; System.out.println("First element value: " + a[0]); System.out.println("The finally statement is executed"); This will produce the following result
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3 First element value: 6 The finally statement is executed Note the following
A catch clause cannot exist without a try statement. It is not compulsory to have finally clauses whenever a try/catch block is present. The try block cannot be present without either catch clause or finally clause. Any code cannot be present in between the try, catch, finally blocks. The try-with-resources Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block. In the following program, we are reading data from a file using FileReader and we are closing it using finally block.
import java.io.File; import java.io.FileReader; import java.io.IOException;
public class ReadData$_{Demo}$
public static void main(String args[]) FileReader fr = null; try File file = new File("file.txt"); fr = new FileReader(file); char [] a = new char[50]; fr.read(a); // reads the content to the array for(char c : a) System.out.print(c); // prints the characters one by one catch(IOException e) e.printStackTrace(); finally try fr.close(); catch(IOException ex) ex.printStackTrace(); try-with-resources, also referred as automatic resource management, is a new exception handling

mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block.
To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block. Following is the syntax of try-with-resources statement.
Syntax
try(FileReader fr = new FileReader("file path")) // use the resource catch() // body of catch Following is the program that reads the data in a file using try-with-resources statement.
Example
import java.io.FileReader; import java.io.IOException;
public class Try$_withDemo$
public static void main(String args[]) try(FileReader fr = new FileReader("E://file.txt")) char [] a = new char[50]; fr.read(a); // reads the contentto the array for(char c : a) System.out.print(c); // prints the characters one by one catch(IOException e) e.printStackTrace(); Following points are to be kept in mind while working with try-with-resources statement.
To use a class with try-with-resources statement it should implement AutoCloseable interface and the close() method of it gets invoked automatically at runtime. You can declare more than one class in try-with-resources statement. While you declare multiple classes in the try block of try-with-resources statement these classes are closed in reverse order. Except the declaration of resources within the parenthesis everything is the same as normal try/catch block of a try block. The resource declared in try gets instantiated just before the start of the try-block. The resource declared at the try block is implicitly declared as final. User-defined Exceptions You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes
All exceptions must be a child of Throwable. If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class. If you want to write a runtime exception, you need to extend the RuntimeException class. We can define our own Exception class as below
class MyException extends Exception You just need to extend the predefined Exception class to create your own Exception. These are considered to be checked exceptions. The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.
Example
// File Name InsufficientFundsException.java import java.io.*;
public class InsufficientFundsException extends Exception private double amount;
public InsufficientFundsException(double amount) this.amount = amount;
public double getAmount() return amount; To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.
// File Name CheckingAccount.java import java.io.*;
public class CheckingAccount private double balance; private int number;
public CheckingAccount(int number) this.number = number;
public void deposit(double amount) balance += amount;

public void withdraw(double amount) throws InsufficientFundsException if(amount <= balance) balance -= amount; else double needs = amount - balance; throw new InsufficientFundsException(needs);
public double getBalance() return balance;
public int getNumber() return number; The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.
// File Name BankDemo.java public class BankDemo
public static void main(String [] args) CheckingAccount c = new CheckingAccount(101); System.out.println("Depositing 500...”); *c.deposit*(500.00);
try System.out.println("100...”); *c.withdraw*(100.00); *System.out.println*(”600..."); c.withdraw(600.00); catch(InsufficientFundsException e) System.out.println("Sorry, but you are short ”+*e.getAmount*()); *e.printStackTrace*(); *CompilealltheabovethreefilesandrunBankDemo*
Output
Depositing 500...
Withdrawing 100...
Withdrawing 600...*Sorry*, *butyouareshort*200.0 InsufficientFundsException at CheckingAccount.withdraw(CheckingAccount.java:25) at BankDemo.main(BankDemo.java:13)
Common Exceptions In Java, it is possible to define two catergories of Exceptions and Errors.
JVM Exceptions These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException. Programmatic Exceptions These exceptions are thrown explicitly by the application or the API programmers. Examples: IllegalArgumentException, IllegalStateException. Suggested Readings "Java Exceptions". 2016. www.Tutorialspoint.Com. https://www.tutorialspoint.com/java/java$_e$*xceptions.htm*.

4.3 Logging

Log4j log4j is a reliable, fast and flexible logging framework (APIs) written in Java, which is distributed under the Apache Software License. log4j is a popular logging package written in Java. log4j has been ported to the C, C++, C, Perl, Python, Ruby, and Eiffel languages.
log4j is highly configurable through external configuration files at runtime. It views the logging process in terms of levels of priorities and offers mechanisms to direct logging information to a great variety of destinations, such as a database, file, console, UNIX Syslog, etc.
log4j has three main components:
loggers: Responsible for capturing logging information. appenders: Responsible for publishing logging information to various preferred destinations. layouts: Responsible for formatting logging information in different styles. log4j features It is thread-safe. It is optimized for speed. It is based on a named logger hierarchy. It supports multiple output appenders per logger. It supports internationalization. It is not restricted to a predefined set of facilities. Logging behavior can be set at runtime using a configuration file. It is designed to handle Java Exceptions from the start. It uses multiple levels, namely ALL, TRACE, DEBUG, INFO, WARN, ERROR and FATAL. The format of the log output can be easily changed by extending the Layout class. The target of the log output

as well as the writing strategy can be altered by implementations of the Appender interface. It is fail-stop. However, although it certainly strives to ensure delivery, log4j does not guarantee that each log statement will be delivered to its destination. Example Step 1: Add log4j dependency to your build.gradle file compile group: 'log4j', name: 'log4j', version: '1.2.17' Step 2: Add log configuration in main/resources/log4j.property
Set root logger level to DEBUG and its only appender to A1. log4j.rootLogger=DEBUG, A1
A1 is set to be a ConsoleAppender. log4j.appender.A1=org.apache.log4j.ConsoleAppender
A1 uses PatternLayout. log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=
Print only messages of level WARN or above in the package com.foo. log4j.logger.com.foo=WARN
Here is another configuration file that uses multiple appenders:
log4j.rootLogger=debug, stdout, R
log4j.appender.stdout=org.apache.log4j.ConsoleAppender log4j.appender.stdout.layout=org.apache.log4j.Pat
Pattern to output the caller's file name and line number. log4j.appender.stdout.layout.ConversionPattern=
log4j.appender.R=org.apache.log4j.RollingFileAppender log4j.appender.R.File=example.log
log4j.appender.R.MaxFileSize=100KB Keep one backup file log4j.appender.R.MaxBackupIndex=1
log4j.appender.R.layout=org.apache.log4j.PatternLayout log4j.appender.R.layout.ConversionPattern=Step 3: Sample log4j program
package logging;
import org.apache.log4j.Logger;
public class LoggingDemo public static void main(String[] args) final Logger logger = Logger.getLogger(LoggingDemo.class); logger.debug("debug statement"); logger.info("info statement"); logger.error("error statement"); Output DEBUG [main] (LoggingDemo.java:10) - debug statement INFO [main] (LoggingDemo.java:11) - info statement ERROR [main] (LoggingDemo.java:12) - error statement Suggested Readings "Log4j Tutorial". 2016. www.tutorialspoint.com. http://www.tutorialspoint.com/log4j/. "Java Logging". 2016. tutorials.jenkov.com. http://tutorials.jenkov.com/java-logging/index.html.

4.4 IDE

Java: IDE IntellIJ 1. Project Manager 2. Search Replace 3. Navigation 4. Formatting 5. Debugging 6. Build Release 7. Git Integration 1. Project Manager 1.1 Create New Project
1.2 Import Maven Project
https://www.jetbrains.com/help/idea/2016.1/importing-project-from-maven-model.html
2. Search Replace Global Search Shift Shift 3. Navigation Next/Previous Error F2 / Shift + F2 4. Formatting Auto Format Ctrl + Alt + L

4.5 Package Manager

Java: Package Manager Gradle
Create your first project with gradle Step 1: Create new project folder
mkdir gradle$_sampleStep2: Makefolderstructure$
gradle init –type java-library Step 3: Import to IntelliJ

Open IntelliJ, click File > New... > Project From Existing Sources... Plugins
Application plugin Usages
1. Using the application plugin
Add this line in build.gradle
apply plugin: 'application' 2. Configure the application main class
mainClassName = "org.gradle.sample.Main"

4.6 Build Tool

Java: Build Tool Apache Ant
Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications. Ant can also be used effectively to build non Java applications, for instance C or C++ applications. More generally, Ant can be used to pilot any type of process which can be described in terms of targets and tasks. 1
Install Ant Download and extract Apache Ant 1.9.6
wget http://mirrors.viethosting.vn/apache//ant/binaries/apache-ant-1.9.6-bin.tar.gz
tar -xzf apache-ant-1.9.6-bin.tar.gz Set path to ant folder
Build Ant through proxy Requirement: 1.9.5+
Add the following lines into build.xml
<target name="ivy-init" depends="ivy-proxy, ivy-probe-antlib, ivy-init-antlib" description="−> initialise Ivy settings"> <ivy:settings file="$ivy.dir/ivysettings.xml"/></target><targetname="ivy-proxy"description="-->ProxyIvysettings"><propertyname="proxy.host"value="proxy.com"/><propertyname="proxy.port"value="8080"/><propertyname="proxy.user"value="user"/><propertyname="proxy.password"value="password"/><setproxyproxyhost=$"proxy.host" proxyport="$proxy.port"proxyuser=$"proxy.user" proxypassword="$proxy.password"/></target>ApacheAnt$

4.7 Production

Java: Production (Docker) Production with java
Base Image: [java]/java
Docker Folder
your-app/ app bin your$_app.shlibDockerfilerun.shDockerfile$
FROM java:7
COPY run.sh run.sh run.sh
cd /app/bin chmod u+x your$_app.sh./your_app.shCompose$
service: build: ./your$_appcommand:'bashrun.sh'$

Tài liệu tham khảo