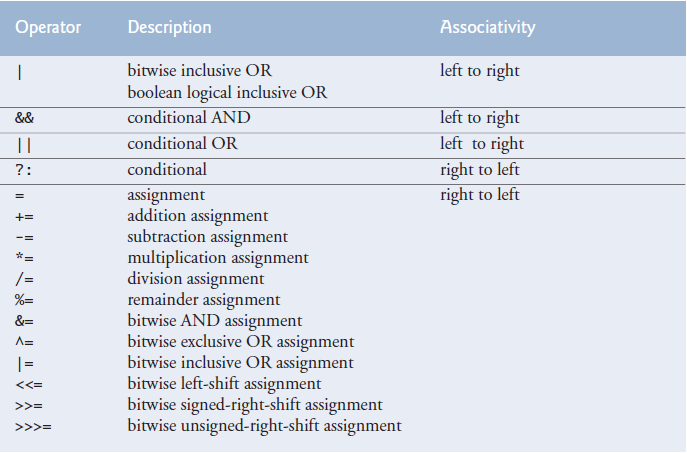
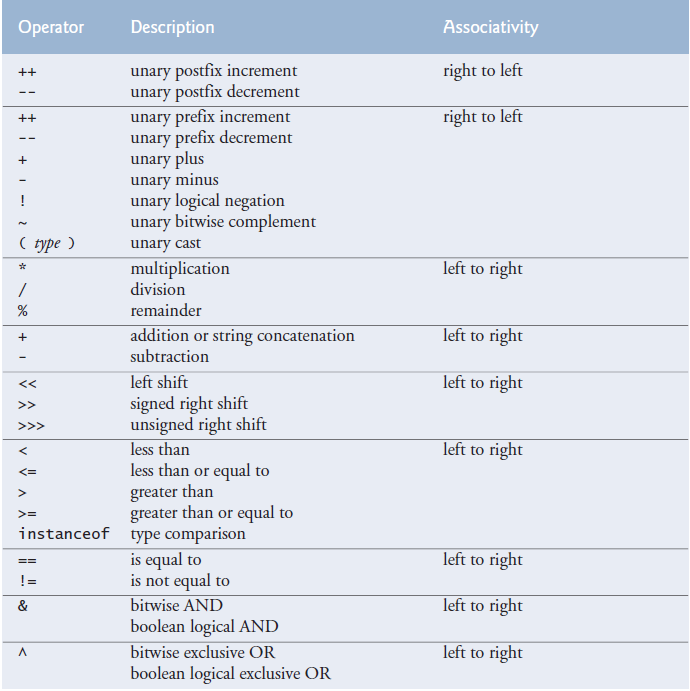
**Part 1**

* **Operators**

(Operator precedence high to low)  
 **-Operators based on operand number:**

**Unary operator:** Contains one operand: “++a;”.

**Binary operator:** Contains two operands: “a = b;”.

**Ternary operator:** Contains three operands: “a > b ? a = 0 : a = 1;”.

Unlike binary operators, the unary operators should be placed next to their operands, with no intervening spaces.

-Unary minus

x = 16 - - 5 -> 16 - (-5) -> x = 16 + 5

x = 16 --5; // Error

a = b + c; // First b + c is calculated and stored in a temporary place then it is assigned to a.

We can use “x % y” and “x / y” to get rid of/gather certain digits. More information on “Programming Tricks & Algorithms” file.  
  
Remainder operator can be used with non-integer operands. But it’s unpopular.

The equality operator “==” should be read as “is equal to” and the assignment operator as “gets the value of” or “gets” to avoid confusion.  
  
You can use parentheses to force the order of evaluation.   
  
**-Bitwise operators** Add what each one does after studying the web appendice.

public static void main() {

int a = 60; // 60 = 0011 1100

int b = 13; // 13 = 0000 1101

int c = 0;

c = a & b; // 12 = 0000 1100

System.out.println("a & b = " + c);

c = a | b; // 61 = 0011 1101

System.out.println("a | b = " + c);

c = a ^ b; // 49 = 0011 0001

System.out.println("a ^ b = " + c);

c = ~a; //-61 = 1100 0011

System.out.println("~a = " + c);

c = a << 2; // 240 = 1111 0000

System.out.println("a << 2 = " + c);

c = a >> 2; // 15 = 0000 1111

System.out.println("a >> 2 = " + c);

c = a >>> 2; // 15 = 0000 1111

System.out.println("a >>> 2 = " + c);

}

**-Shorthand operators:** A shorthand operator is a shorter way to express a statement that is already available in a language.

**-Compound/Composite assignment operators:** += -= \*= /= %= operators are compound operators. They are short versions of original statements as shown below. (tag: compound assignment operators)

variable = variable operator expression;

variable operator= expression;

x = x + 2;

x += 2;

-Compound operators have built-in cast operators:

int i = 5;

long j = 8;

i = i + j; // Will not compile

i += j; // Will compile

because

i += j;

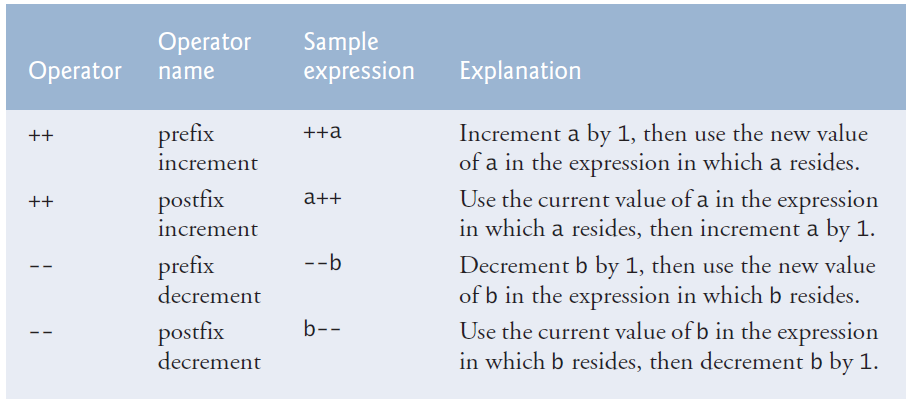
stands for

i = (int)(i + j);

=+ vs +=  
  
 x =+ 5 -> x = (+5)  
  
 x += 5 -> x = x + 5

**-Increment and decrement operators:** Used to add or subtract 1 from the value of a numeric variable.

Attempting to use the increment or decrement operator on an expression other than one to which a value can be assigned is a syntax error. For example, “++(x + 1)” is a syntax error, because “(x + 1)” is not a variable, it is an expression.



Using the prefix increment (or decrement) operator to add 1 to (or subtract 1 from) a variable is known as preincrementing(or predecrementing). This causes the variable to be incremented (or decremented) by 1; then the new value of the variable is used in the expression in which it appears. Using the postfix increment (or decrement) operator to add 1 to (or subtract 1 from) a variable is known as postincrementing(or postdecrementing). This causes the current value of the variable to be used in the expression in which it appears; then the variable’s value is incremented (or decremented) by 1.

z = x++ + y;

When incrementing or decrementing a variable in a statement by itself, the prefix increment and postfix increment forms have the *same* effect, and the prefix decrement and postfix decrement forms have the *same* effect. It’s only when a variable appears in the context of a larger expression that preincrementing and postincrementing the variable have different effects (and similarly for predecrementing and postdecrementing).

-Three different ways of incrementing/decrementing

passes = passes + 1;

passes += 1;

++passes;

passes++;

x = x++; // Does nothing

* **Data Types**

The table lists the eight primitive types in Java. Like its predecessor languages C and C++, Java requires types of all variables to be explicitly declared and thus determined at compile time. For this reason, Java is referred to as a statically typed language. In Java, variables are bound to specific data types, and if types do not match up as expected in the expression, it will result in type errors. For this reason, Java is referred to as a strongly typed language.

-Portability: The primitive types in Java are portable across all computer platforms that support Java.

-Size of bool:Size of primitive bool variable changes. It can be 8 or 16 or more bits. If you want bool variables that are one bit big then you need to use bool arrays. [Class BitSet](https://docs.oracle.com/javase/7/docs/api/java/util/BitSet.html)

A variable exceeding its max or min limit is known as arithmetic overflow and causes undefined behavior.  
 It is easy to comprehend overflow in integers. When you are calculating floating-point values, the number can get so small that it overflows and becomes negative. That’s why when you are calculating e, after a while it starts to subtract instead of adding.  
  
 Capture

-Scientific notation:Can be used with a float or double.

double a = 1e-3;

float b = 1e+3F;  
  
Float variables areSingle-precision (32 bits) floating-point numbers. Can hold up to 7 significant digits. Double variables are double-precision (64 bits) floating-point numbers. Double requires twice the memory but can hold 15 significant digits. Float and double can hold real numbers, not just rational numbers. [Real Numbers](https://image01.ipracticemath.com/content/imageslm/realnumber/real_number.png)

Floating-point numbers are not %100 precise. We have float, double and BigDecimal depending on our needs.

Maximum and minimum values that can be stored in an int variable are represented by constants MIN\_VALUE and MAX\_VALUE, which are defined in class Integer. There are similar constants for the other integral and floating-point types. Each primitive type has a corresponding class type in package java.lang.

char can represent letters, digits, special characters (\*, %), escape sequences (\n). chars are held as integers in the computer.

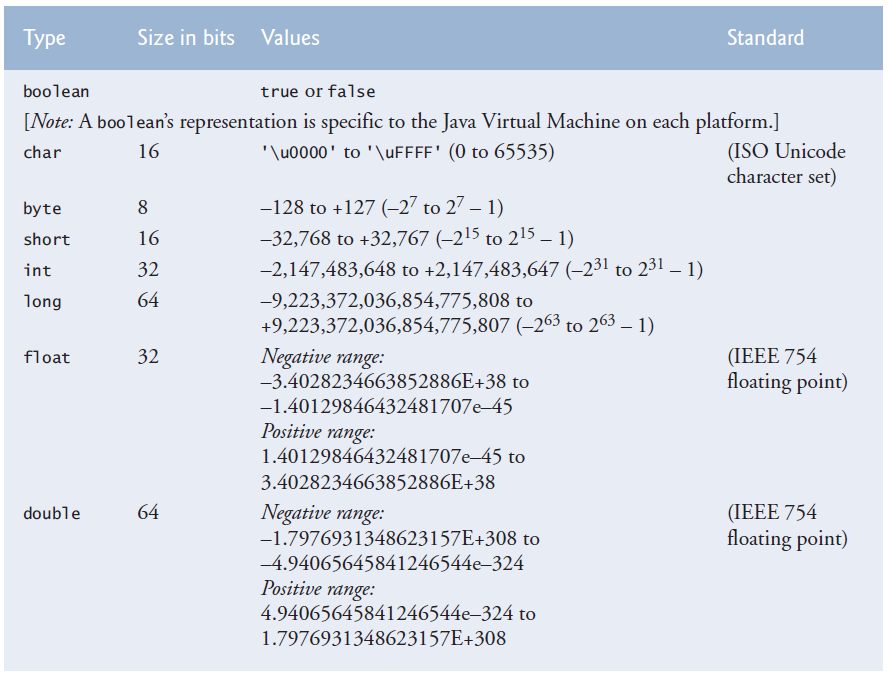
Java uses UTF-16.

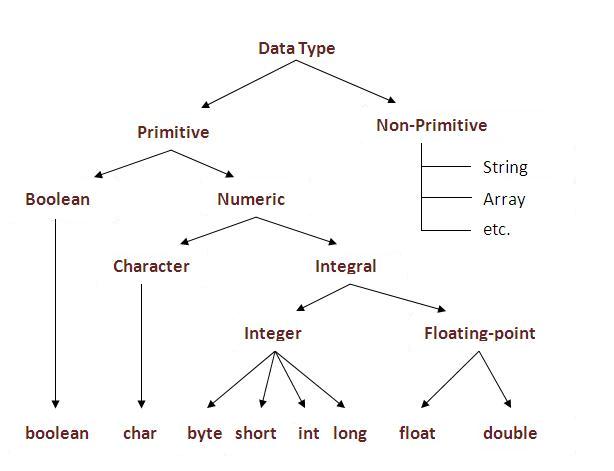
Most Java program text consists of [ASCII](https://en.wikipedia.org/wiki/ASCII) characters, but any Unicode character can be used as part of identifier names, in comments, and in character and string literals. For example, π (Greek lowercase letter pi) is a valid Java identifier. ASCII character set is a subset of the Unicode character set used by Java.

char unicode1 = 'a';

char unicode2 = 'π';

char unicode3 = '\u2202';





**-Default values:** Default values of primitive type variables and reference type variables.

|  |  |
| --- | --- |
| **Data Type** | **Default Value** |
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0F |
| double | 0.0 |
| char | '\u0000' |
| boolean | false |
| reference | null |

**-Default data types:** When we are using char values we must do it between ‘ ’. When using string we must do it between “ ”. When using long we must use L after it. When using float we put F after it. When using floating-point literals, explicitly use the decimal point to let people know this is supposed to be a floating-point.

In Java, integer literals are considered type int, and floating-point literals are considered type double.

Creating a floating point copy (9.0) of the 9 integer literal and then assigning that to float type myVariable.

float myVariable = 9;

Trying to assign a double type literal to a float type variable, which requires an explicit type casting. The statement won't compile.

float myVariable = 9.5;

Explicitly casting double type literal to a float type literal and assigning it to float type variable.

float myVariable = (float) 9.5;

Assigning float type 9.5 literal to a float type variable.

float myVariable = 9.5F;

Suffixes are only needed for literal values. You don’t need to use them when you are inputting data to a program. Because when you are reading a value from a user you are doing it with methods like nextDouble and nextFloat. When you are reading values with methods like nextFloat(), your program doesn’t reject the 5.0 double value because it uses a casting operator behind the scenes.

**-Data Type Conversions**

**-Explicit conversion / Type Casting**

A cast operator can be used to convert between primitive numeric types, and between related reference types. Casting to the wrong type may cause compilation errors or runtime errors.  
  
You cant make an assignment that would cause data loss in Java. Also, you are not allowed to use two different data types in a calculation. In order to do these, you need to use the cast operator.

Dividing two integers result in integer division, any fractional part of the calculation is truncated (discarded). Integer division doesn’t round. It truncates. So 7/4 won’t result in 1,5 rounding to 2. It will result in 1.5 truncating to 1.

To perform a floating-point calculation with integer values, we must *temporarily* treat these values as floating-point numbers for use in the calculation. Java provides the unary cast operator to accomplish this task. For example (double) cast operator can be used to create a *temporary* floating-point copy of its operand total (which appears to the right of the operator). Using a cast operator in this manner is called explicit conversion or type casting. The value stored in total is still an integer.

average = (double) total / counter

You don’t always have to use the cast operator. In some situations such as using literal values, you can just use float or double values instead of using integer values and casting them to floating-point values.  
  
4.0F / 3.0F instead of (float) 4 / 3  
 4.0 / 3.0 instead of (double) 4 / 3

**-Implicit conversion / Promotion** [argument\_promotion\_and\_casting](#argument_promotion_and_casting)

The calculation now consists of a floating-point value (the temporary double copy of total) divided by the integer counter. Java can evaluate only arithmetic expressions in which the operands’ types are *identical*. To ensure this, Java performs an operation called promotion (or implicit conversion) on selected operands. For example, in an expression containing int and double values, the int values are promoted to double values for use in the expression. Then floating-point division is performed and the result of the calculation is assigned to average. As long as the (double) cast operator is applied to *any* variable in the calculation, the calculation will yield a double result.

Argument promotion is done in expressions and method argument-parameter type matching.

-float i = 13;

-int c = 12;

someMethod(c);

void someMethod(double c1) {

…  
}  
  
The result of an expression involving anything int sized or smaller is always int. This is because   
  
byte b1 = 12;  
byte b2 = 13;  
byte b3 = b1 + b2; // Compiler error. Possible lossy conversion from int to byte.  
byte b4 = 21 + 12; // Compiles, implicit cast. Why does this work?

**­-String <=> Primitive**

String stringNumber = Integer.toString(number);

String stringNumber = String.valueOf(number); // Returns null instead of throwing an exception when reference doesn’t point to anything.

int numberString = Integer.parseInt(stringNumber);

int numberString = Integer.valueOf(stringNumber);

**-Unicode to char and String**

short code = 2202;

char c1 = (char) code;

String unicodeString = Character.toString(c1);

**-Base**

-Integers literals in a different base: The table below just shows how you can write numbers in bases other than 10. They don’t convert integer values to other types of values. You can precede them with a minus (-) to make them negative.

// Decimal declaration and possible chars are [0-9]

int decimal = 495;

// Hexadecimal declaration starts with 0X or 0x and possible chars are [0-9, A-F, a-f]

int hexa = 0x1EF;

// Octal declaration starts with 0 and possible chars are [0-7]

int octal = 0757;

// Binary representation starts with 0B or 0b and possible chars are [0-1]

int binary = 0b111101111;

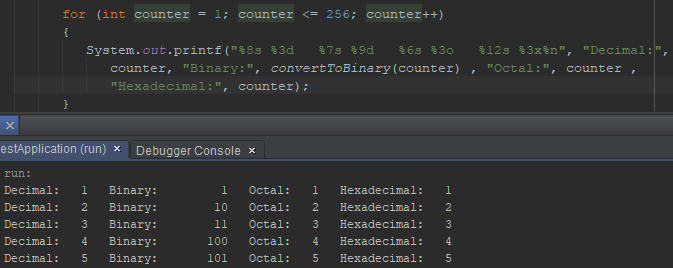
-Outputting integers in a different base:You can convert a number to any base you want. There are two toString methods in primitive data type classes. It can take only a value or a value and a radix. It can be any radix, not just 2, 8, and 16.

Integer.toString(x, 2); // Binary

Integer.toString(x, 8); // Octal

Integer.toString(x, 16); // Hexadecimal

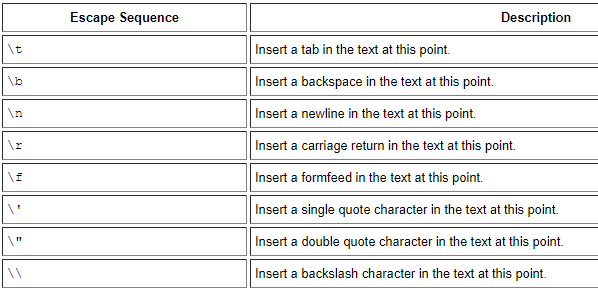
Or you can use conversion characters.



-Octal and hexadecimal chars:

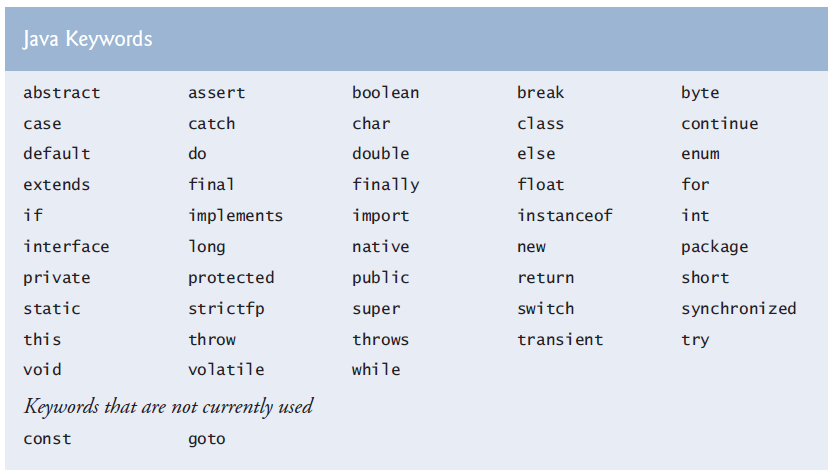
char c = '\ooo'; // Octal 101 is decimal 65, which is the character ‘A’.

char x1 = '\uhhhh'; // Hexadecimal 41 is decimal 65, which is the character ‘A’.

**-Control characters, escape characters, escape sequences:** “\” is the escape character. \n is an escape sequence. When a backslash appears in a string (variable, literal, format string), Java combines it with the next character to form an escape sequence.  
  


**Part 2**

* **Keywords and Reserved Words:** Reserved for use by Java and are always spelled with all lowercase letters.

  
Java also contains the reserved words true and false, which are boolean literals, and null, which is the literal that represents a reference to nothing. Like keywords, these reserved words cannot be used as identifiers.

Hard coded values in source code that stand for a fixed amount are literals. For example, 3.75 is a floating-point literal. 3 is an integer literal.

**-Native:** The native keyword is applied to a method to indicate that the method is implemented in native code using JNI (Java Native Interface).

* **Programming Style / Conventions**

Arithmetic expressions in Java must be written in straight-line form.

Java is case sensitive. Java commands are case sensitive too.

**-Identifier:** The names we give to packages, classes, interfaces, methods, and variables to access them easily instead of using their address in memory. Identifiers consist of letters, digits, underscores, and dollar signs, and they don’t begin with a digit (so the compiler understands it’s not a number) and don’t contain spaces. The convention is:

Starting with a letter.

The dollar sign is not used at all.

If you are using more than one word, capitalize the first letter of the words after the first one. This is called camelCase. Also capitalizing the first letter is called PascalCase.

Class and enum identifiers should start with an uppercase letter and should be a noun.  
 Exception-class identifiers should end with the word Exception.  
 Interface identifiers should start with an uppercase letter and should be an adjective.  
 Method identifiers should start with an uppercase letter and should be a verb.  
 Variable identifiers should start with a lowercase letter.  
 Constant variable identifiers should be all uppercase such as “NUM” or “NUM\_GEARS”. This is also the only recommended way of using underscore.

Package identifiers should be all lowercase.   
  
List instance variables first in the class’s body. Also, declare one variable per line. Use good identifiers and comments to make the program readable and understandable.

Setter and getter methods start with words set and get. Predicate method names begin with “is” rather than “get”.  
  
The main method argument is named "args" or "argv".   
  
When using if else structures don’t indent “else if” or “else”.

if (studentGrade >= 90)

System.out.println("A");

else if (studentGrade >= 80)

System.out.println("B");

else

System.out.println("F");

We put a space between control statements and their conditions but we don’t put a space between method names and their argument list.  
  
 if (x > 10)  
  
 factorial(x);  
  
Don’t return a result of a calculation. Calculate it, assign it to a variable, and then return that result. Easier to know what is being returned.

If a single statement must be split across lines, choose natural breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines.

System.out.printf(“Name in object myAccount is: %n%s%n”,

myAccount.getName());  
  
Put one empty line after a block if there is more code under it.

while (…)

{

…

}

// More code

* **Terms**

**-Expression,** **Statement**

Expression:Portions of statements that have values are called expressions. Expressions are formed by combining variables, constants, and method calls, using operators.

Statements instruct the computer to perform actions. The semicolon at the end of a line means the end of the statement has been reached. Statements can span multiple lines. ; is called a statement terminator.

Empty statement (null statement): Can be used with other control statements too.

if (number1 == number2); // This would be interpreted by Java as,  
  
if (number1 == number2)  
; // Empty statement  
  
**-Assignment, Initialization** int x;  
 x = 10; // assignment  
int x = 10; // initializtion  
  
**-White space characters:** Newline (\n), Space, Tab (\t). Ignored by the compiler.

**-Comments: Comments are deleted when the program is compiling.**

-**End of line comment:** // End of line comment

-**Traditional comments:** /\* Traditional comment \*/

**-Boxed comments:**  /\*\*\*\*\*\*\*\*

\* \*

\* \*

\*\*\*\*\*\*\*\*/

-**Javadoc comments** enable you to embed program documentation directly in your programs. This way of embedding documentation is the conventional way of doing it. The Javadoc utility program generates HTML pages based on these comments.

/\*\* documentation \*/

**-Tokens** are the smallest individual building block of a Java program. The Java compiler uses them for constructing expressions and statements. A Java program is a collection of tokens, comments, and white spaces.

Keywords

Identifiers  
 Literals

Operators

Separators

**-Separators:** () {} [] ; , . :

***Part 3***

* **Package**

Premade related classes are grouped into packages and are collectively referred to as Java class library or the Java Application Programming Interface(Java API). To put classes you write to the same package use the keyword package,  
  
package packageName;

**Classes in same directory are in the same package. Classes in same package dont need importing. They are implicitly imported.**

**-Import:** If you look at libraries part of your project, you will see that Java API jars are already added to your project. To add libraries that come with Netbeans or other 3rd party libraries, you right click “Libraries” of your project and click “Add Library” or “Add JAR/folder” respectively. Lastly, maven libraries are kept in “C:\Users\Michael\.m2\repository” and added to your project scope from there. When libraries area added to your project, they become reachable with their fully qualified names. If we want to, we can import these libraries to reach them without using their fully qualified names.

Single-type-import declaration specifies one class to import. (tag: single type import declaration)

import org.junit.Assert;

Type-import-on-demand declaration imports only the classes that the program uses from a particular package. (tag: type import on demand declaration)  
  
import java.util.\*;  
   
We cant use type-import-on-demand declaration for multiple packages in a folder,

import java.\*;

Fully qualified class name: You dont need to import classes if you are willing to type their full package name and class name every time.

Java.util.Scanner input = new java.util.Scanner(System.in);

importingjava.lang.System is not needed because classes in this package are automatically imported. Because they are the most essential classes to run java programs. The predecessor of all classes (Object class) is inside this package too.

Classes of java.lang are not the only implicitly imported classes. Classes in the same package are also implicitly imported.   
  
Importing classes from default package is a compile time error. You cant import a class from there because that package has no name.

API of old Java versions are called deprecated and might be removed from never versions. The compiler will warn you when you compile code that uses deprecated APIs. If you compile with javac using command line argument –deprecated, the compiler will tell you which deprecated features you are using.

**-Static import:** [static\_import](#static_import)

* **Introduction to Classes**

Every Java program consists of at least one class. Keyword class introduces a class declaration.

If we have a public class in a .java file, that .java file must be named after that public class “ClassName.java”. This is the only public class in a .java file because there can only be one public class in a .java file.

Driver class:When we have a class without main method, we need other class(es) that have main methods to use these classes from. Those classes are called driver classes. (tag: client of the class, client of a class)

* **Introduction to Methods**

Methods perform tasks and can return information when they complete their tasks.The parentheses after an identifier indicate it’s a method. Main is a method too and it has arguments too. The method header is the first line of a method where you specify method’s return type, name and parameter list (parameter types and names). Calling a method is also referred to as invokinga method. Methods are generally public. Most methods must be called on a specific object.

objectName.methodName(arguments)

Arguments and return values dont have to be primitive data types. They can be objects, enum constants, etc. Also arguments and return values may be constants, variables or expressions.

-private enum Coin { HEADS, TAILS };

public static Coin flip()

-public static void displayAccount(Account account1) {  
System.out.printf("%s balance: $%.2f%n", account1.getName(),

account1.getBalance());  
}  
  
Account.displayAccount(account1);

**-Main method** is the starting point of every Java application (It is always called automatically by JVM when you execute an application). There must be a main method in a Java application (class with main method) and it must be defined as “public static void main(String[] args)”, otherwise the JVM will not execute the application. Main method is public and static so JVM can access it and access it without instantiating an object of the class. Program terminates when it reaches the end of main method. Main class/application class is the class that has the main method.

A .java file doesnt have to have a class with main method. But it needs a main method to be executable by itself.   
 Main method doesn’t have to be in public class.   
 We can have multiple classes in a .java file, only one of them can be public, main method can be in all of them. One class can have multiple main methods but the main method we use with Java command executes. You can overloaded main methods, that is you can change number or type of parameters which will be considered as a different method by the compiler since it has a different sign. Depending on the main method arguments, the correct main method will be chosen.   
 Even though you wrote the classes in the same .java file, all classes have their own .class file. So the main method of the .class file that is being executed, is the one that is used.

Main method can have a return type. It can return an int by using “System.exit(int);” or “return int;” when we are at root main method. Operating system interprets return value. 0 for succesful exit, 1 for unsuccesful exit.  
  
Static methods and variables can be used without creating an object of the class.

When you are using a method inside it’s class, you dont have to specify what class. You just need to type the method’s name.  
  
Setter and getter methods are used to set and get the values of instance variables safely. A get method that returns a boolean value is commonly called a predicate method.  
  
public void setState(String state) {  
 this.state = state;  
}  
  
public String getState() {  
return state;  
}

public boolean isNoFaultState() {  
boolean noFaultState;

switch (getState()) {

case "MA": case "NJ": case "NY": case "PA":

noFaultState = true;

break;

default:

noFaultState = false;

break;  
}  
return noFaultState;  
}

* **Console Input/Output**

**-System.out** object is predefined and known as the standart output object/stream. “out” is a static reference variable in System class. It references to a PrintStream object. [streams part2](#streams_part2) [Class System](https://docs.oracle.com/javase/7/docs/api/java/lang/System.html)

public static final PrintStream out = null; // This code is in class System.

**-print, println, printf:** One string(argument to print methods) can’t span more than one line of code(all of the string has to be in one line) but you can use multiple strings in a print to divide a string in a line to multiple lines in code and concatenate those strings.

Print method displays a line of text.

Prntln puts a newline after printing.

Printf(print formatted): We can print formatted without printf(using +). Or we can use printf.

Printf’s first argument is a format string that may consist of fixed text, format specifiersandcontrol characters. Each format specifier is a placeholder for a value. We can also do the math inside printf (By using expressions instead of values/variables in non first arguments).

Outputs that lead user to take an action are called prompt.

Newline Character and the line seperator:%n (a format specifier, the line seperator) only works with printf and is portable. \n (newline character) can be used with all three print methods but is not portable across OS, file processing, networking, etc. Println uses %n implicitly.

Unix systems use newlines (\n) to mark line endings in text files. The MacOS uses carriage-returns (\r). Windows uses a carriage-return followed by a newline (\r\n). [Newline](https://en.wikipedia.org/wiki/Newline)

System.out.println("Welcome to Java Programming!"); // If you use a string   
 // literal, it is automatically converted to a String object by the compiler.

System.out.print("Welcome to ");

System.out.println("Java Programming!");

System.out.println(“Welcome to \nJava Programming!”);

System.out.printf("Welcome to " // String concatenation

+ "Java Programming!");

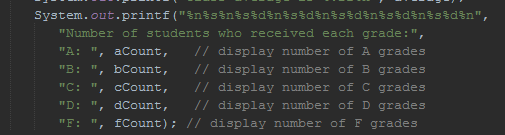
System.out.printf(“%s%n%s%n”, // These two rows are one statement.

“Welcome to ”, “Java Programming!”);

System.out.printf("%s %d%n%s %d%n", "Welcome to", aCount,

"Java Programming!", bCount);

System.out.printf(“Sum is %d%n”, number1 + number2);



We put ‘+’ to sides of variables/values where there is a string when we want to use formatted strings with print and println.

System.out.println(i);

System.*out*.println(name + “ “ + age);

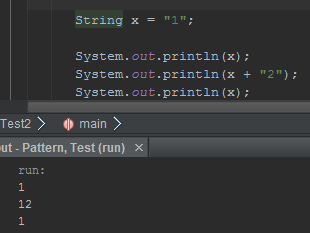
System.*out*.println("cadence: " + cadence + ", speed: " + speed);

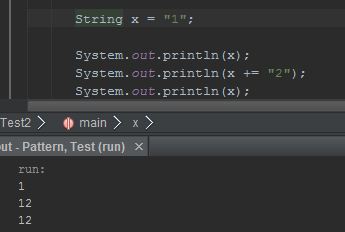
System.out.println("Population in " + year++ + ": " + (population \*= growthRate));

-Formatting big numbers to make them easier to read

You can make big numbers more readable by using underscores. We can use this with values being displayed as binary, octal, hexadecimal too.

for (int roll = 1; roll <= 6\_000\_000; roll++)  
  
 long population = 0x7fff\_ffff\_ffffL;  
  
 System.out.printf("Population in %d: %,d%n", year++, pop);  
  
 System.out.println(String.format("%,d", pop));  
  
 System.out.println(String.format("%,d", pop).replace(",", " "));  
  
**-String concatenation**Java allows you to assemble String objects into larger strings by using operators +, += or concat(String).





+ takes the string or non-string literals/variables in both sides and merges them in a new temporary string that gets collected by garbage collector later. toString method is called implicitly for non string variables(including objects). At least one side has to be string.

+= takes the string or non-string literal/variable on the right side and adds it to the string variable on the left side.

If there are any trailing zeros in a double value, these will be discarded when the number is converted to a String—for example 9.3500 would be represented as 9.35.

A boolean concatenated with a String is converted to the String "true" or "false".

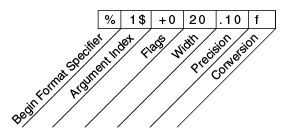
Confusing the + operator used for string concatenation with the + operator used for addition

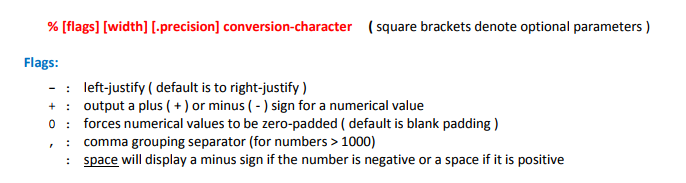
can lead to strange results. Java evaluates the operands of “addition or string concatenation” operator, from left to right. For example, if integer variable y has the value 5, the expression

"y + 2 = " + y + 2

results in the string "y + 2 = 52", not "y + 2 = 7", because first the value of y (5) is concatenated to the string "y + 2 = ", then the value 2 is concatenated to the new larger string "y + 2 = 5". The expression "y + 2 = " + (y + 2) produces the result "y + 2 = 7".

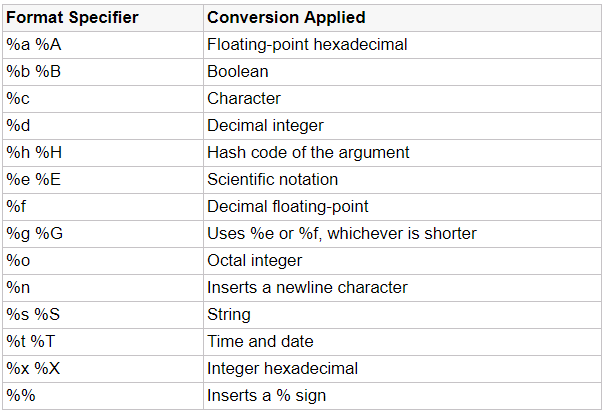
**-Formatting Output**-Format specifier

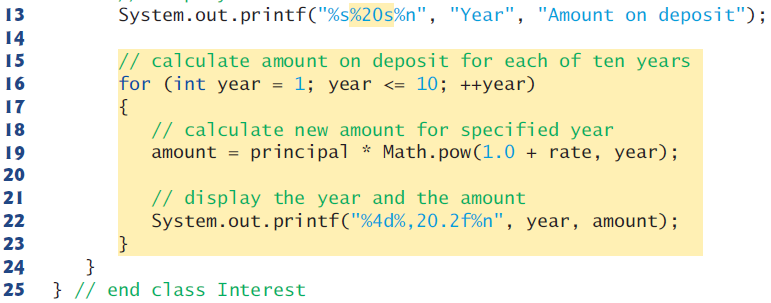
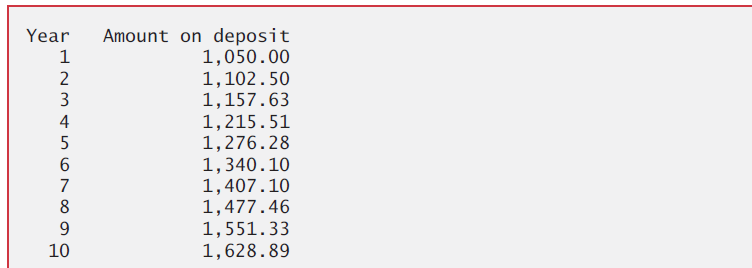
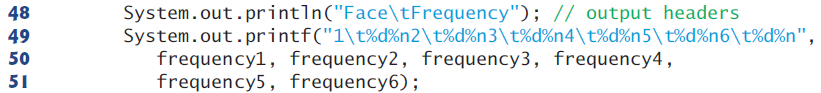
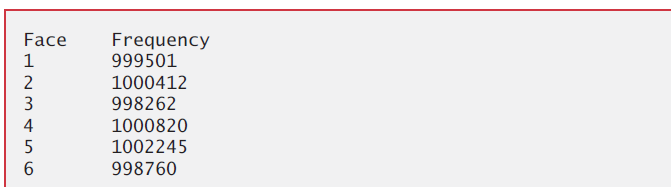




(tag: minus flag, plus flag, zero flag, comma flag, space flag, blank flag)

-Conversion characters



-Printing a table: Print the first item of the table header. To allign first column of other rows use width of first column of table header. In our example it’s four.  
  
  
  
  
  
  
  


The integer 20 between the % and the conversion character s indicates that the value should be displayed with a field widthof 20—that is, printf displays the value with at least 20 character positions. If the value to be output is less than 20 character positions wide, the value is right justifiedin the field by default. If the year value to be output were more than four character positions wide, the field width would be extended to the right to accommodate the entire value—this would push the amount field to the right, upsetting the neat columns of our tabular output. To output values left justified, simply precede the field width with the minus flag (–)(e.g., %-20s).

%,20.2f. The comma flag (,)indicates that the floating-point value should be output with a grouping separator. The actual separator used is specific to the user’s locale (i.e., country). For example, in the United States, the number will be output using commas to separate

every three digits and a decimal point to separate the fractional part of the number, as in

“1,234.45”. The number 20 in the format specification indicates that the value should be

output right justified in a *field width* of 20 characters. The .2 specifies the formatted number’s

*precision*—in this case, the number is *rounded* to the nearest hundredth and output

with two digits to the right of the decimal point. By default, floating-point values are displayed with six digits precision. (tag: default precision)

Floating point values can be showed to look like integers by printing them with no digits after floating point(%.0f). But this is not recommended both when you are printing and when you are writing source code. Users and devs should know it’s a floating point value.  
  
-Dynamic width and precision part 3: [dynamic precision part2](file:///C:\Users\Michael\AppData\Roaming\Microsoft\Word\-%20Other\Java%20API%20Notes.docx#dynamic_precision_part2) Unfortunately we dont have the %\*.\*f feature from C.  
  
 String.format(String.format("%%.%ds", precision), "foo")

-A Warning about Displaying Rounded Values: Here’s a simple explanation of what can go wrong when using double (or float) to represent dollar amounts, assuming that dollar amounts are displayed with two digits to the right of the decimal point: Two double dollar amounts stored in the machine could be 14.234 (which would normally be rounded to 14.23 for display purposes) and 18.673 (which would normally be rounded to 18.67 for display purposes). When these amounts are added, they produce the internal sum 32.907, which would normally be rounded to 32.91 for display purposes. Thus, your output could appear as



but a person adding the individual numbers as displayed would expect the sum to be

32.90.

Do not use variables of type double (or float) to perform precise monetary calculations. The imprecision of floating-point numbers can lead to errors. Java provides class java.math.BigDecimal for this purpose.  
  
**-System.in** object is predefined and known as the standart input object/stream. “in” is a static reference variable in System class. It references to an InputStream object. [streams part1](#streams_part1) [streams part3](#streams_part3)

public static final InputStream in = null; // This code is in class System.

**-Scanner** (java.util.Scanner): System.in allows program to read byte sized data inputted by user. The “next…” methods turn those byte sized data into Java data types.

You should check if the value entered is in type you want and in format/range you want. Telling user what they need to enter clearly, validating, range checking, re-asking when wrong value is entered are important parts of programming.

-**nextLine:** Pressing enter puts a newline character after the characters you typed. When you press enter characters are read by the method until it encounters newline and returns a string containing character up to but not including newline, which is discarded.

-Extra newline character: NextLine reads the value you type and the newline character. Non NextLine readers dont absorb the newline character. So if you used a non NextLine, you need to absorb its newline character with a next or nextLine method. And after that you can read a text with next or nextLine.

You dont need to get rid of the extra newline character when you are reading the next int or double etc. because they ignore the newline character and only read what they are looking for.

// First way, absorb the newline character with a nextLine method.

int x = input.nextInt();

input.nextLine();

String s = input.nextLine();

// Second way, do the assigning, check if Scanner object has been emptied or   
 // check if your string has changed.

int x = input.nextInt();  
 String s = input.nextLine();

if(input.hasNextLine() || s.isEmpty()){

s = input.nextLine();

}

-**next:** Reads the next word. When you press enter after typing some text, method next reads characters until it encounters a white space character, then returns a string containing characters up to, but not including the white space character which is discarded. Only white space character is disarded, you can continue reading rest later.

You dont have to assign a value to a variable to use that value. You can just use a value you get from the user directly.

if (input.nextInt() == 0)

setName(input.nextLine());

System.out.println(input.nextLine());

**-EOF(End Of File):** C works with EOF and newline character and other characters much more since C works with characters. Java works with lines/words so you dont “keep reading characters until EOF (or null), newline character, whitespace character is found”. Instead you, “as long as there are more lines or words, keep printing them”.  
  
The end-of-file indicator (EOF)is a system-dependent keystroke combination which the user enters to indicate that there’s *no more data to input*. On UNIX/Linux/Mac OS X systems, end-of-file is entered by typing the sequence is Ctrl + D on a line by itself. On Windows systems, end-of-file can be entered by typing Ctrl + Z. On some systems, you must press *Enter* after typing the end-of-file key sequence. Also, Windows typically displays the characters ^Z on the screen when the end-of-file indicator is typed.

**-hasNext:** Scanner method hasNext determines whether there’s more input data. This method returns the boolean value true if there’s more data; otherwise, it returns false. As long as the end-of-file indicator has not been typed, method hasNext will return true.

But this method does not consume entered value. So in the example below, when we type a value the hasNext method just sees if it is EOF or not. If it is EOF, the while statement ends. If it is not EOF, than the value you enter can be read by another method. In this case it is nextInt method.

while (input.hasNext()) {

int grade = input.nextInt();

total += grade;

}

while (input.hasNext()) {

System.out.println("counter: " + counter + ", input: " + input.nextLine());  
 counter++;

}

**-**has methods of Scanner class returns boolean values. The Scanner objects looks for the next token in their inputs and they return true or false depending on whether the next token is what you are looking for. There are different has methods to see if the next token is a certain type or not. hasNext is for seeing if there is more input or not(any type).

* **Introduction to Object Oriented Programming**

**-Creating Classes and objects:** Creating an object of a class is called instantiation. Objects are instances of a class. Each new class you create becomes a new type that can be used to declare variables and create objects.

Account: type,

account1: reference variable,

the object referenced by the reference variable account1: object

You can declare new classes to fit your needs whenever you want. Thats why Java is an extensible language.

**­ -New:** The keyword new allocates memory for the object (Requests for space from the system.). The constructor to the right of keyword new which creates the object is called implicitly by the keyword new. Address of the created object(result of right value, which is an expression) is assigned to left value (which is a reference variable with a name and a type specified) using the assign operator. “new Account()” is called “class instance creation expression”.

Account account1 = new Account(“Jane Green”);

**-Variable Types**

A variable is alocation in computer’s memory where a value can be stored for use later in a program. All java variables must be declared with a name and type before they can be used. All variables have a name, type, size and a value. A variable’s name enables the program to access the value of the variable in memory. Destructive process is a process that causes some data to be lost(writing). Non destructive process is a process that doesn’t cause data to be lost(reading).

**-Primitive types:** Primitive types are int, boolean, byte, char, short, long, float and double. All non primitive types are reference types. A primitive type can hold one value at a time.

**-Reference types:** Reference type variables(called references) store adresses of objects in computer’s memory. Such a variable is said to refer to an object in the program. To call methods of an object, you need a reference to the object(myAccount.getName).

Size of reference variables is generally 32 bit at 32 bit systems and 64 bit in 64 bit systems.

referenceVariable = null; // It doesnt point to anything anymore.

**-Garbage Collector:** Every object uses system resources such as memory. We need a disciplined way to give resources back to the system when they’re no longer needed; otherwise, “resource leaks” might occur that would prevent resources from being reused by your program or possibly by other programs. The JVM performs automatic garbage collection to reclaim the memory occupied by objects that are no longer used. When there are no more references to an object, the object is eligible to be collected. Collection typically occurs when the JVM executes its garbage collector, which may not happen for a while, or even at all before a program terminates. So, memory leaks that are common in other languages like C and C++ (because memory is not automatically reclaimed in those languages) are less likely in Java, but some can still happen in subtle ways. Resource leaks other than memory leaks can also occur. For example, an app may open a file on disk to modify its contents—if the app does not close the file, it must terminate before any other app can use the file.

So close your streams, files and other I/O classes when you are done using them.

input.close(); // Closing Scanner object.

If any objects are not reclaimed before the program terminates, the operating system will reclaim the memory used by the program. When it does, it’s possible that no objects or only a subset of the eligible objects will be collected.

-Finalize:Never use the method finalize of Class object. The original intent of finalize was to allow the garbage collector to perform termination housekeeping on an object just before reclaiming the object’s memory. Now, it’s considered better practice for any class that uses system resources—such as files on disk—to provide a method that programmers can call to release resources when they’re no longer needed in a program. We talk more about this type of allocating and deallocating resources in chapter 11 (deallocating resources in finally block, using try with resources).

input.close();

**-Field** **(Instance Variable, Class Variable)**

Declared inside class but outside methods.

Its scope/visibility depends on its adornment.

They are used to represent attributes of a class.

You dont have to initialize them. They have default values. Class loader and JIT compiler initializes fields to their default values, not constructors. You can use constructors to assign value during object creation.

Fields exists as long as the object exists.

Most instance variable declarations are preceded with private keyword so the variable can only be used by methods of its own class.

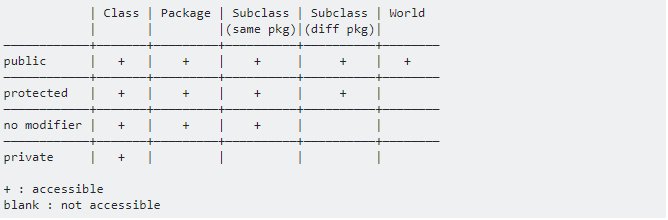
class A {  
 int hour;  
   
 void setTime(int h) {  
 hour = h;  
 }  
 }  
 **-Local Variable**  
  
 Declared inside a block (method, constructor, or control structure).  
 Can only be used in their block they were declared in(they can be used from the line they are declared until end of their block. Each method(including constructors and main) can access only its local variables. Control structures can use local variables that are declared outside them in the same method/constructor. But local variables declared inside control structures can’t be used outside them in the same method/constructor.  
 They are used to store information(state) temporarily.   
 You have to initialize them. They don’t automatically initialize to their default value.  
 Local variables exists until their scope ends.  
  
 **-Argument:** Arguments are the values we use while invoking a method. These values are passed to the variables in method called parameters.   
  
 **-Parameter:** They are declared in the parameter list. They are initialized with arguments. Argument number has to be equal to parameter number. Types dont have to be same type as we will see later but they have to be compatible/consistent [argument\_promotion\_and\_casting](#argument_promotion_and_casting). You have to specify a type and a name for each parameter.   
  
**-Adornments / Access modifiers:** Determines which scopes can access classes, methods and fields.

**Public(+):** Can be reached from all packages in the project.

**Protected(#):** Can be reached from own class, package, other package subclass.

**no modifier:** Can be reached from own class, package.

**Private(-):** Can be reached from own class.

  
  
World: Other packages (directories).

No modifier is also called package private or package access.

Public methods are also called public services or public interface. Private methods are also called utility methods or helper methods.

-Adornment of classes can be public or package private. Only nested classes can be also protected or private.  
-Adornment of Instance variables and methods can be any of the four access modifiers. Generally instance variables are private. So you cant access them directly from outside class. You need to use public methods of that object to access these variables.  
  
- myAccount.name = "test"; // If name is public you can access it from other classes  
  
- String theName = input.nextLine();  
 myAccount.setName(theName); // This is how you access private instance variables.  
  
**-Scope:** Scope refers to the lifetime and accessibility of a variable.

**1.** The scope of a parameter declaration is the body of the method in which the declaration

appears.

**2.** The scope of a local-variable declaration is from the point at which the declaration

appears to the end of that block.

**3.** The scope of a local-variable declaration that appears in the initialization section

of a for statement’s header is the body of the for statement and the other expressions

in the header.

**4.** A member’s scope is the entire body of the class. But this scope can be extended depending on the access modifier. [Accessing Methods and Fields](#accessing_methods_and_fields)

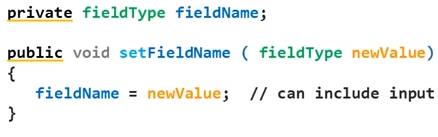
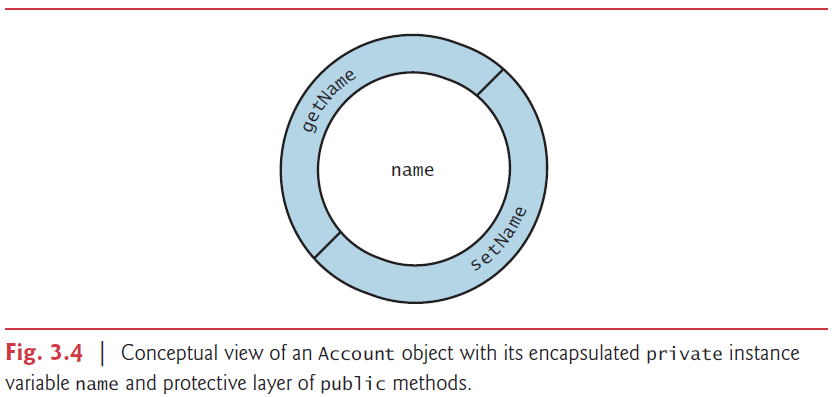
-Principle of Least Privilege:In a general sense, “things” should have the capabilities they need to get their job done, but no more. An example is the scope of a variable. A variable should not be visible when it’s not needed. Or if the variable doesn’t need to change after initialization we should make it final.

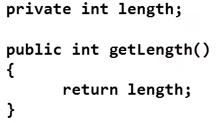
**-Setter, Getter**

-Safety of set and get methods:It isnot safe to give direct access to instance fields. You might think set and get methods dont change anything security wise since you still read or write anything you want, just through methods. But set and get methods are supposed to have controls that let you only read and write certain types of values in certain ways. This makes our code more robust and strong. [getters\_setters\_part2](#getters_setters_part2) [getters\_setters\_part3](#getters_setters_part3)Declaring instance variables with private access modifier is known as data hiding or information hiding.  
  
When a program creates an object of a class, instance variables are encapsulated (hidden, and related data and methods are bundled)in the object and can be accesses only by methods of object’s class.  
  
 **-Setter**

Commonly called mutator methods.

We are be able to directly assign values to instance/class variables when they are not private.

Usually setter methods have void return type, use exception handling to prevent and indicate attempts to assign invalid data.   
  
 station1.location = miami;  
  
   
  
 public void deposit(double depositAmount) {  
 if (depositAmount > 0,0)  
 balance = balance +depositAmount;  
 }  
  
  
(tag: *encapsulation*, data hiding, information hiding)  
 **-Getter**  
  
Commonly called accessor methods or query methods.

Use “return this.variableName” if variableName is shadowed.  
  
   
  
**-Static:** [static\_part2](#static_part2)Static fields and methods are shared among the class and its objects. Static members exist and can be used even when no objects of the class have been instantiated. They are available as soon as the class is loaded into memory at execution time. (static main method is an example to this.)  
  
-Static methods: Sometimes a method performs a task that does not depend on an object. Such a method applies to the class in which it’s declared as a whole and is known as a static method or a class method. Static methods and variables are generally used for common fields and tasks that doesnt need an object.

For any class imported into your program, you can call the class’s static methods by specifying the name of the class in which the method is declared, followed by a dot (.) and the method name,

ClassName.methodName()

You can’t use “this” to refer to a static method.

Non-static methods are typically called instance methods.

Static methods in a class can call each other without a need for an object. We can even call the main method but we dont.

Static methods are implicitly final. You can hide (different from shadowing) the parent static method by just making a brand new static method with the same signature. When you want to use the static method of the parent class, you dont use super.methodName. You use ClassName. methodName. keywords “this” and “super” refer to objects. [static\_binding2](#static_binding2) (tag: hidden)

-Static variables: There are variables for which each object of a class does *not* need its own separate copy. Such variables are declared static and are also known as static variables or class variables. When objects of a class containing static variables are created, all the objects of that class share *one* copy of those variables. Non-static variables are called instance variables. Together a class’s static variables and instance variables are known as its fields. Also, together a class’s methods and fields are known as its members.

For example E and PI variables in Math class are declared with the modifiers public, final and static.

ClassName.variableName

Using static variables instead of giving the same instance variable to all objects saves time and memory and is error resistant.

Using static methods doesnt save time and memory because Java holds only one copy of instance methods anyway. [single method in memory part1](#single_method_in_memory_part1) [single\_method\_in\_memory\_part2](#single_method_in_memory_part2)

-Static Object/Static Class: Static object in Java is actually a class (It doesnt make sense to make an object static anyway). Static means you can access it without creating an object of the class. So the real static things are methods and fields. Also you prevent the class from being instantiated, all of whose members are declared static. It can be used in an application to provide global access to methods or variables.  
  
To have a static class, make class final, make constructor private, make all members static.

public class TestMyStaticClass {

public static void main(String[] args) {

MyStaticClass.setMyStaticMember(5);

System.out.println("Static value: " + MyStaticClass.getMyStaticMember());

System.out.println("Value squared: " +   
 MyStaticClass.getSquareMyStaticMember());

// MyStaticClass x = new MyStaticClass(); // results in compile time error

}

}

// A top-level Java class mimicking static class behavior

final class MyStaticClass {

private MyStaticClass () { }

private static int myStaticMember;

public static void setMyStaticMember(int val) {

myStaticMember = val;

}

public static int getMyStaticMember() {

return myStaticMember;

}

public static int getSquareMyStaticMember() {

return myStaticMember \* myStaticMember;

}

}

Nested classes can be actually static. Instantiating static nested class from static context, [nested\_class\_part2](#nested_class_part2) [Static classes](https://stackoverflow.com/questions/7486012/static-classes-in-java) [Static object](https://www.quora.com/What-is-the-purpose-of-a-static-object-in-Java-When-is-it-actually-used-or-in-which-context) [Singleton pattern](http://www.techrepublic.com/blog/software-engineer/using-the-singleton-pattern-in-java/)  
  
  OuterClass.NestedClass myNested = new OuterClass.NestedClass();

Instantiating inner class from static context.  
  
 OuterClass oc = new OuterClass();

OuterClass.InnerClass ic = oc.new InnerClass();

-Static initializer: We initialize instance variables with constructors. But we cant call a constructor without creating an object. So in order to initialize class variables conditionally we use static blocks.

Static initialization block runs after the inline static initializers (after initializing static fields).

Static initializers can initialize static final fields too.

public class Test {

static Scanner input = new Scanner(System.in);

static final int B;

static int H;

static boolean flag = true;

static

{

B = input.nextInt();

input.nextLine();

H = input.nextInt();

input.close();

if ((B < 0) || (H < 0))

{

flag = false;

System.out.println("java.lang.Exception: Breadth and height must be   
 positive");

}

}

public static void main(String[] args)

{

if (flag)

{

int area = B \* H;

System.out.print(area);

}

}

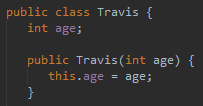
}

[Static initialization block](https://stackoverflow.com/questions/9379426/java-when-is-a-static-initialization-block-useful/49639856#49639856) [Initializers vs constructors](https://stackoverflow.com/questions/804589/use-of-initializers-vs-constructors-in-java)

**-This, variable shadowing:** It is sometimes called this reference.You canuse the instance variable directly if there is no local variable with the same name. If there is a local variable with the same name, the field gets shadowed. So you need to use “this.fieldName” to access the field.

Every object can access a reference to itself with keyword this. When an instance method or constructor is called for a particular object, the method’s or constructor’s body implicitly uses keyword this to refer to the object’s instance variables and other methods. This enables the class’ code to know which object should be manipulated.

We cant use this reference when we are in a static context.

  
  
First use of this reference is “this.variableName” and “this.methodName()” are used to access instance variables and instance methods of an object.  
  
Second use of this reference is, accessing constructors using this().

**-Constructor:** Each class you create can optionally provide constructors with parameters that can be used to initialize its objects.

Java requires a constructor call for every object that is created. Even if you didnt create a constructor, there is an implicit constructor provided by compiler. If you declare a constructor yourself then there is no default constructor. If there is no default constructor that means you cant create an object with default constructor “new Account()” unless you create one yourself.

Constructors cant have return types, not even void. They are generally public. They cant be called, they are automatically called when you create an object.

public Account(String name) {

this.name = name;

}

public Account(String name, int age) { // Constructors can be overloaded

this.name = name;

this.age = age;

}

Constructors must have the same name as the class. A constructor will have a parameter list. When you create a new object with parameters of a specific constructor, that constructor will execute, the arguments you entered will be passed to constructor’s parameters and from there it will be used to initialize instance variables etc.

Java allows other methods of the class besides its constructors to have the same name as the class and to specify return types. Such methods are not constructors and will not be called when an object of the class is instantiated.

You can call methods from constructors but you shouldnt. Because fields are not initialized yet. This can cause problems.

* **Logical operators**

**-Condition** is an expression that can be true or false. In Java, your condition expressions must be boolean.

Things you can use as a condition:

true

false

A boolean type variable

A simple condition (2 operands and either a relational operator or an equality operator) or a complex condition (simple conditions merged using && or || or ^).

if, if…else, while, do…while, for and switch statements each require a condition to determine how to continue a program’s flow of control. Simple conditions are expressed in terms of the relational operators >, <, >= and <= and the equality operators == and !=, and each expression tests only one condition. To test multiple conditions in the process of making a decision, we perform these tests in separate statements or in nested if or if…else statements. Sometimes control statements require more complex conditions to determine a program’s flow of control.

Java’s logical operatorsenable you to form more complex conditions by combining

simple conditions. The logical operators are && (conditional AND), || (conditional OR), &

(boolean logical AND), | (boolean logical inclusive OR), ^ (boolean logical exclusive OR)

and ! (logical NOT).

The & (bitwise AND), | (bitwise inclusive OR) and ^ (bitwise exclusive OR) operators are bitwise operators when they’re applied to integral operands.

**-Conditional Operators**

**-Conditional AND(&&) Operator:** Complex condition is true when both simple conditions are true.

if (gender == FEMALE && age >= 65)

++seniorFemales;

Some programmers find that the preceding combined condition is more readable when *redundant* parentheses are added, as in:

if((gender == FEMALE) && (age >= 65))

**-Conditional OR(||) Operator:** Complex condition is true when*either or both* simpleconditions are true.

if ((semesterAverage >= 90) || (finalExam >= 90))

System.out.println ("Student grade is A");

-Short Circuit evaluation of complex conditions: The parts of an expression containing && or || operators are evaluated *only* until it’s known whether the condition is true or false. Thus, evaluation of the expression

(gender == FEMALE) && (age >= 65)

stops immediately if gender *is not* equal to FEMALE (i.e., the entire expression is false) and

continues if gender *is* equal to FEMALE (i.e., the entire expression could still be true if the

condition age >= 65 is true). This feature of conditional AND and conditional OR expressions

are called short-circuit evaluation.

Problem with this feature is if we have a statement such as

if (a < b) && (++c < 10)

the c variable wont be incremented if first condition is false. Such dense coding style is not recommended. “++c” expression is called a side effect. Using “boolean and” or “boolean or” will prevent this error. Or we can use dependent condition.

In expressions using operator &&, a condition—we’ll call this the dependent condition— may require another condition to be true for the evaluation of the dependent condition to be meaningful. In this case, the dependent condition should be placed after the && operator to

prevent errors. Consider the expression (i != 0) && (10 / i == 2). The dependent condition

(10 / i == 2) must appear after the && operator to prevent the possibility of division by zero.

**-Boolean Logical AND (&) and Boolean Logical Inclusive OR (|) Operators:** The & and | operators are identical to the && and || operators, except that the & and | operators always evaluate both of their operands (i.e., they do not perform short-circuit evaluation).

This is useful if the right operand has a required side effect.

Dont use side effect in your code.

Assignment (=) expressions generally should not be used in conditions. Every condition must result in a boolean value; otherwise, a compilation error occurs. In a condition, an assignment will compile only if a boolean expression is assigned to a boolean variable.

**-Boolean Logical Exclusive OR (^):** A simple condition containing the ^ operator is true if and only if one of its operands is true and the other is false. If both are true or both are false, the entire condition is false.

This operator is guaranteed to evaluate both of its operands.

**-Logical Negation (!) Operator:** The !(logical NOT, also called logical negationor logical complement) operator “reverses” the meaning of a condition. Unlike the logical operators &&, ||, &, | and ^, which are *binary*operators, the logical negation operator is a *unary* operator.

You can use ! to reverse the value of boolean type expressions/variables.

if (!boolean1)

if (!(grade == sentinelValue))

System.out.printf("The next grade is %d%n", grade);

In most cases, you can avoid using logical negation by expressing the condition differently

with an appropriate relational or equality operator. For example, the previous statement may also be written as follows:

if (grade != sentinelValue)

System.out.printf("The next grade is %d%n", grade);

-De morgan theorem: This theorem state that the expression !(condition1 && condition2) is logically equivalent to the expression (!condition1 || !condition2).

* **Algorithms and Pseudocode**

[Algorithm Notes.docx](file:///C:\Computer%20Engineering\3-%20Algorithm,%20Data%20Structures\Algorithm%20Analysis\3-%20Notes\Algorithm%20Notes.docx) (tag: top down stepwise refinement,bottom up)

* **Control Statements and Structured Programming**

Normally statements in a program are executed one after the other in the order in which they are written. This process is called sequential execution. Using various statements to specify the next statement to execute is called transfer of control.

The term Control structures comes from the field of computer science. *The Java Language Specification* refers to “control structures” as “control statements”.

During the 1960s, it became clear that the random use of transfers of control was the root of much difficulty experienced by software development groups. The blame was pointed at the goto statement (used in most programming languages of the time), which allows you to specify a transfer of control to one of a wide range of destinations in a program. [Note: Java does not have a goto statement; however, the word goto is reserved by Java and should not be used as an identifier in programs.]

The research of Bohm and Jacopini had demonstrated that programs could be written without any goto statements. The challenge of the era for programmers was to shift their styles to “goto-less programming.” The term structured programming became almost synonymous with “goto elimination.” Not until the 1970s did most programmers start taking structured programming seriously. The results were impressive. Software development groups reported shorter development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes was that structured programs were clearer, easier to debug and modify, and more likely to be bug free in the first place.

Bohm and Jacopini’s work demonstrated that all programs could be written in terms of only three control structures; the sequence structure, the selection structure and the repetition structure.   
 Controlling expression is the expression that gives the value to control statement. Control statement works differently depending on that value.

**-Sequence Structure:** The sequence structure is built into Java and other modern programming languages. Unless directed otherwise, the computer executes Java statements one after the other in the order in which they’re written—that is, in sequence. Java lets you have as many actions as you want in a sequence structure. Anywhere a single action may be placed, we may place several actions in sequence.

-**Selection Statements:** Java has three types of selection statements The *if statement* either performs an action if a condition is *true*, or skips it if the condition is *false*. The *if*…*else statement* performs an action if a condition is *true* and performs a different action if the condition is *false*. The *switch statement* performs one of *many* different actions, depending on the value of an expression.

The if statement is a single-selection statement because it selects or ignores a *single* action (or *group of actions*). The if…else statement is called a double-selection statement because it selects between *two different actions* (or *group of actions*). The switch statement is called a multiple-selection statement because it selects among many different actions (or group of actions).  
  
In fact, it’s straightforward to prove that the simple if statement is sufficient to provide *any* form of selection—everything that can be done with the if…else statement and the switch statement can be implemented by combining if statements (although perhaps not as clearly and efficiently).

**-Repetition Statements:** Java provides three repetition statements (also called iteration statements or looping statements) that enable programs to perform statements repeatedly as long as a condition (called the loop-continuation condition) remains *true*. The repetition statements are the while, do…while, for and enhanced for statements(enhanced for is a different way of using the for statement). The while and for statements perform the action (or group of actions) in their bodies zero or more times—if the loop-continuation condition is initially *false*, the action (or group of actions) will *not* execute. The do…while statement performs the action (or group of actions) in its body *one or more* times.

-Calculations: In loops, avoid calculations for which the result never changes—such calculations should typically be placed before the loop. Many of today’s sophisticated optimizing compilers will

place such calculations outside loops in the compiled code.  
  
It’s straightforward to prove that the while statement is sufficient to provide *any* form of repetition. Everything that can be done with do…while and for can be done with the while statement (although perhaps not as conveniently).

**-Counter controlled repetition:** We have a counter(control variable) to control the number of times a set of statements will execute. Counter controlled repetition is often called definite repetition because the number of repetition is known before the loop begins executing.

-Elements of counter controlled repetition: Control variable, initial value, increment, loop continuation condition.

Floating point variables are approximate so you shouldnt use them as counters. Use integers instead.

This code saves a statement, Coding in such a condensed fashion takes practice, might make code more difficult to read, debug, modify and maintain, and typically should be avoided.

int counter = 1;

while (counter++ <= 10) // loop-continuation condition

System.out.printf("%d ", counter);

**-Sentinel controlled repetition:** In definite repetition we knew how many times the loop was going to execute. But in this case we dont know how many times the loop is going to execute. The loop is going to execute until sentinel value is entered.

This repetition uses a special value called a sentinel value(also called a signal value, sentinel, a dummy value or a flag value) to indicate “end of data entry.” The user enters grades until all legitimate grades have been entered. The user then types the sentinel value to indicate that no more grades will be entered. Sentinel-controlled repetition is often called indefinite repetition because of the number of repetitions is not known before the loop begins executing.

Clearly a sentinel value must be chosen that cannot be confused with an acceptable input value.

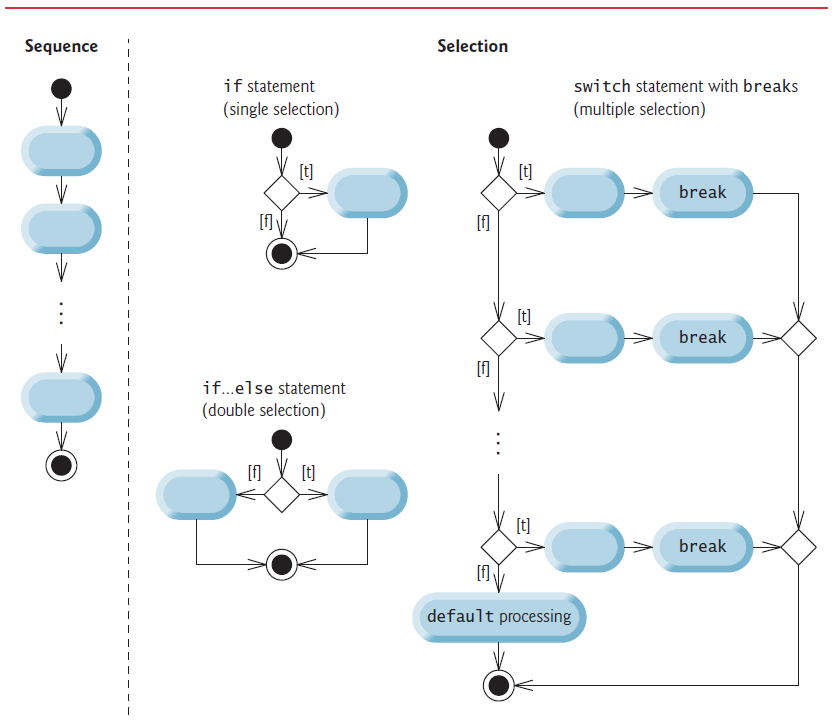
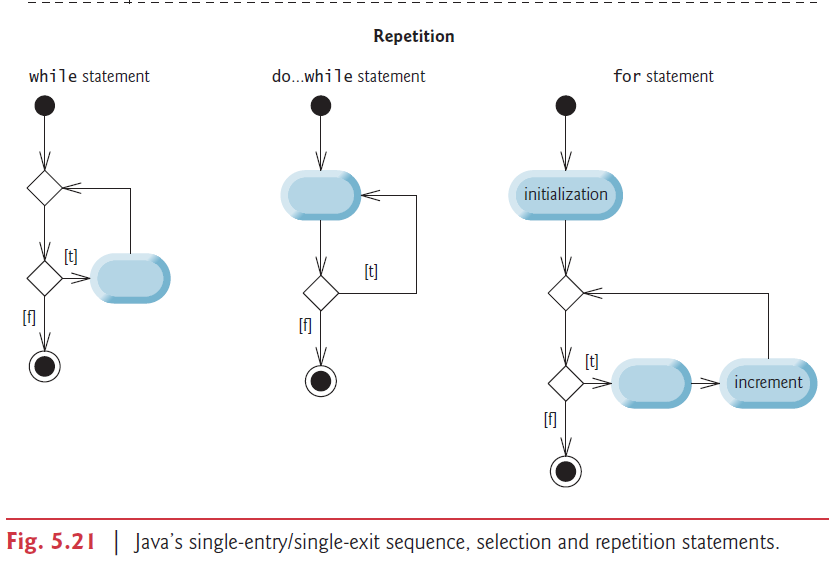
In sentinel controlled loops you should prompt the sentinel value to user.

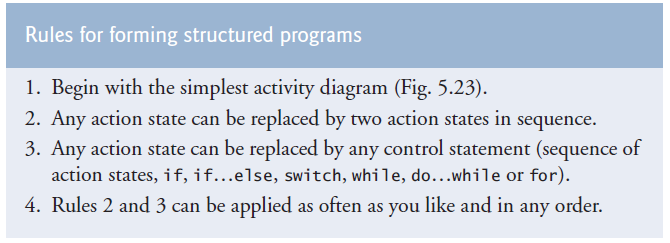
-**Structured programming summary:** Java has only three kinds of control statements: the sequence statement, selection statements (three types), repetition statements (three types). Every program is formed by combining as many of these statements as is appropriate for the algorithm the program implements.

For simplicity, Java includes only single-entry/single-exit control statements—there’s only one way to enter and only one way to exit each control statement. (black circle and black circle with black ring around it). (tag: single entry, single exit)

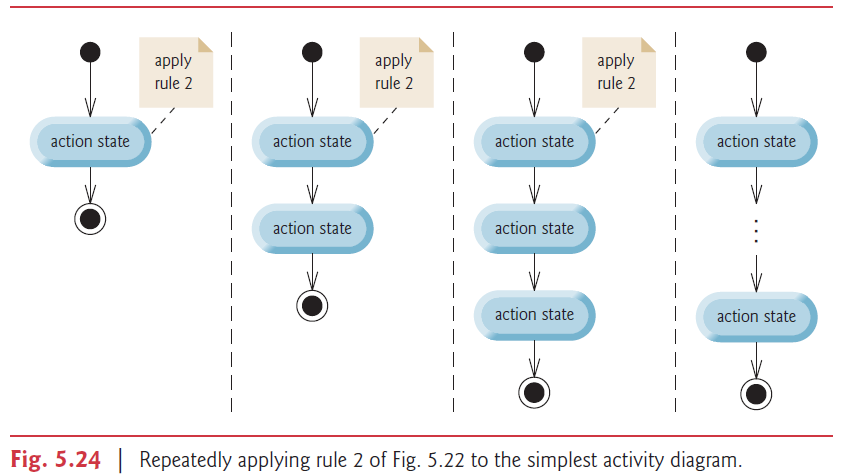
Connecting the exit point of one statement to the entry point of the next is called control statement stacking. When a control statement is inside another, we call that control statement nesting.

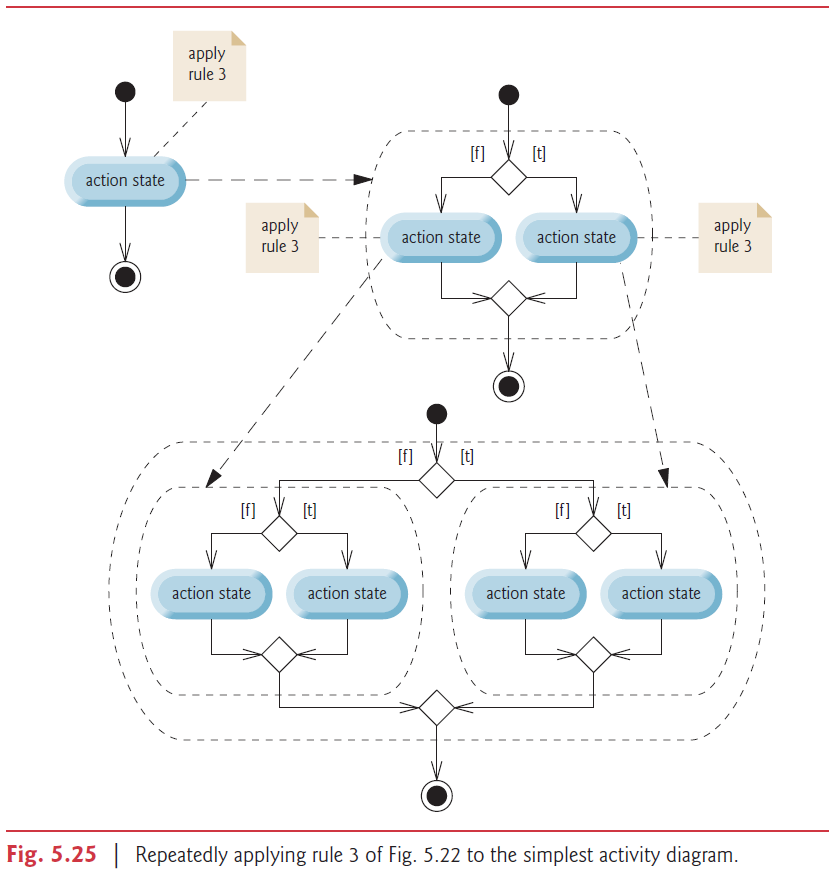
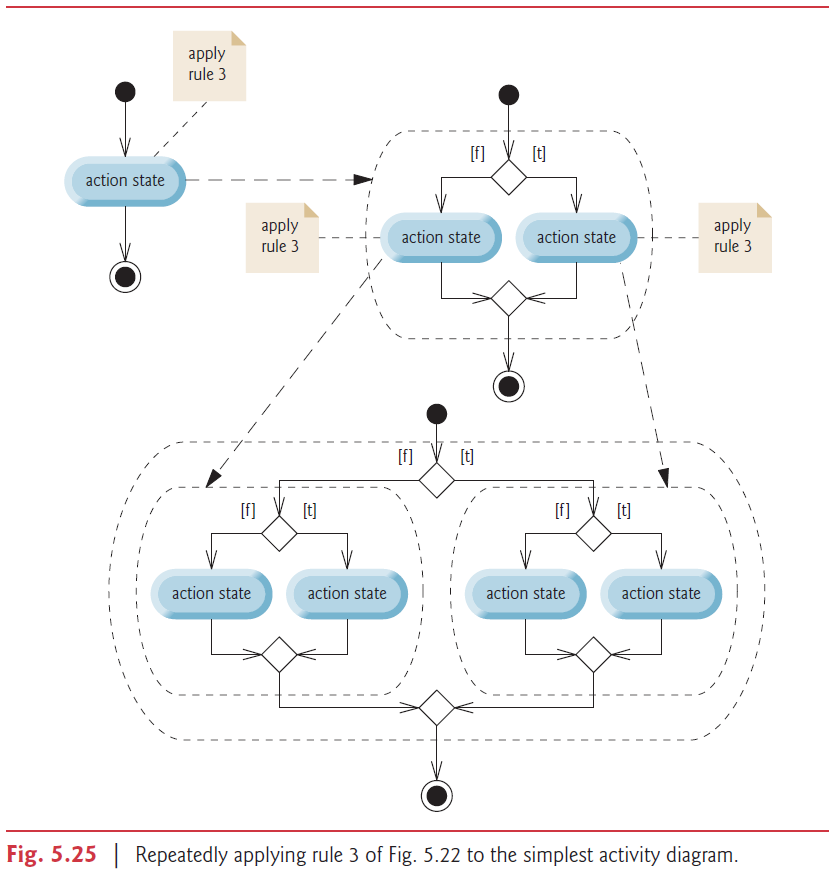
Thus, algorithms in Java programs are constructed from only tree kinds of control statements, combined in only two ways. This is the essence of simplicity.



Applying the rules always results in a properly structured activity diagram with a neat, building-block appearance. For example, repeatedly applying rule 2 to the simplest activity diagram results in an activity diagram containing many action states in sequence. Rule 2 generates a *stack* of control statements, so let’s call rule 2 the stacking rule.



Rule 3 is called the nesting rule. Repeatedly applying rule 3 to the simplest activity diagram results in one with neatly *nested* control statements. For example, in Fig. 5.25, the action state in the simplest activity diagram is replaced with a double-selection (if…else) statement. Then rule 3 is applied again to the action states in the double-selection statement, replacing each with a double-selection statement.  
  
  
  


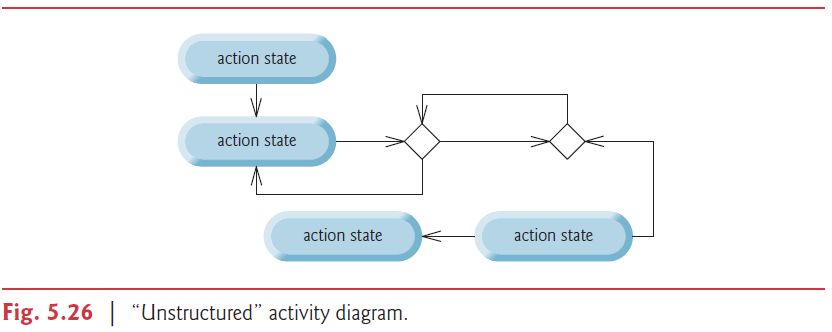
Rule 4 generates larger, more involved and more deeply nested statements. The diagrams that emerge from applying the rules constitute the set of all possible structured activity diagrams and hence the set of all possible structured programs. The beauty of the structured approach is that we use *only seven* simple single-entry/single-exit control statements and assemble them in *only two* simple ways.

If the rules are followed, an “unstructured’ activity diagram (like the one in Fig. 5.26) cannot be created. If you’re uncertain about whether a particular diagram is structured, apply the rules in reverse to reduce it to the simplest activity diagram. If you can reduce it, the original diagram is structured; otherwise, it’s not.

First thing wrong with this program is that the arrow coming out of second decision symbol goes in the middle of the do while statement. It should go to the beginning if we want two nested do while statements. Going to the middle of a do while statement directly can only be achieved with a goto statement.

Second thing wrong with this program is, there is no way of reaching the second part of the program. There is not even a goto statement that jumps to the second part.

Last wrong thing is the fact that there are two out outgoing transition arrows from a statement.



**-Conditional control Statements** (More in “[UML Notes.docx](file:///C:\Computer%20Engineering\13-%20Software%20Engineering,%20Project%20Development\Software%20Engineering\UML\UML%20Notes.docx)” file.)

-**if**: If statement allows you to perform actions when the condition is true. An if statement expects one statement in its body, but may contain multiple statements if they are enclosed in a set of braces “{}” . Indentation is optional but recommended because it emphasizes the inherent structure of structured programs.

**-if else:** If else performs actions when the condition is true and perform another set of actions when the condition is false.

if (grade >= 60)

System.out.println(“Passed”);

else

System.out.println(“Failed”);

When you are using if else statements make sure you test for all cases.

**-Dangling else problem:** Java compiler associates an else with the immediately preceding if unless told otherwise by the placement of braces. First code below will be read like second code below by compiler. To get rid of the problem we need to write these statements like third code block.

if (x > 5) // first program

if (y > 5)

System.out.println(“x and y are > 5”);

else

System.out.println(“x is <= 5”);

if (x > 5) // second program

if (y > 5)

System.out.println(“x and y are > 5”);

else   
 System.out.println(“x is <= 5”);

if (x > 5) { // third program

if (y > 5)

System.out.println(“x and y are > 5”);

}

else

System.out.println(“x is <= 5”);

**-Blocks:** The Java compiler expects only one statement to be in if’s body. So if you have more than one statement, compiler will think those statements come after the if statement’s end. In order to get around this problem you need to use braces like you do for dangling else problem. Statements contained in a pair of braces form a block.

A block of statements can be placed anywhere in a method that a single statement can be placed. It is possible to have an empty statement anywhere a single statement can be placed too.

-**Conditional operator(?): Can be used instead of if else statement.**

**The statement,**

**statement1 ? statement2 : statement3;**

**means this,**

**if(statement1)**

**statement2**

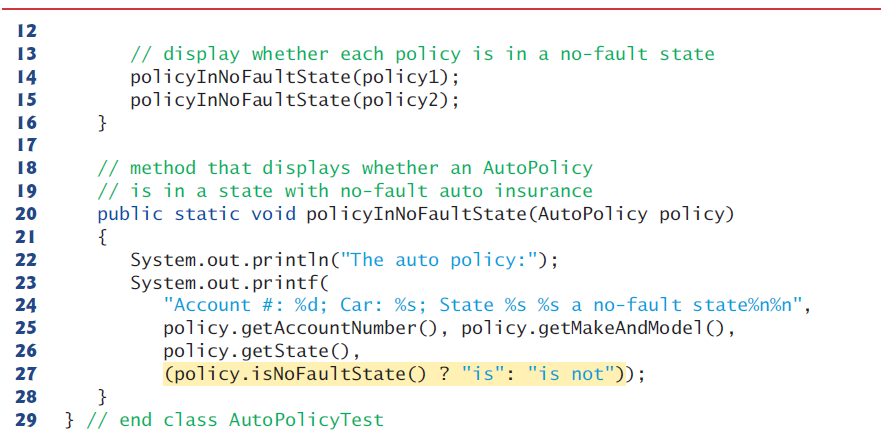
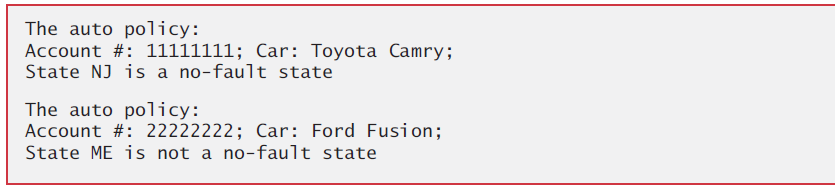
**else**

**statement3**

**Conditional operator is the only ternary operator in Java.**

If statement 2 and 3 are not statements but values, conditional operator returns one of these values depending on the condition. Together, the operands and the ?: symbol form a conditional expression.

result = point >= 50 ? "Successfull" : "Unsuccessfull";  
  
 System.out.printf("%s\n", (count % 2) == 1 ? "\*\*\*\*" : "++++++++");

Use expressions of same type for second and third operands of conditional operator to avoid error caused by wrong variable types.

**-while:** A repetition statement allows you to specify a program should repeat an action while some condition remains true. Both while and for statements can be used for both sentinel and counter controlled repetition. But while is generally used for sentinel controlled and for is generally used for counter controlled statements.

-Counter controlled while statement

int count = 1;

while (count <= 10) {

System.out.println("Count is: " + count);

count++;

}

Not providing statements in the body of a while statement that eventually causes the condition to become false normally results in a logic error called an infinite loop.

We stated that while could be used in most cases in place of for. This is *not* true when the increment expression in the while follows a continue statement. In this case, the increment does *not* execute before the program evaluates the repetition-continuation condition, so the while does not execute in the same manner as the for.

while (count < 11) {

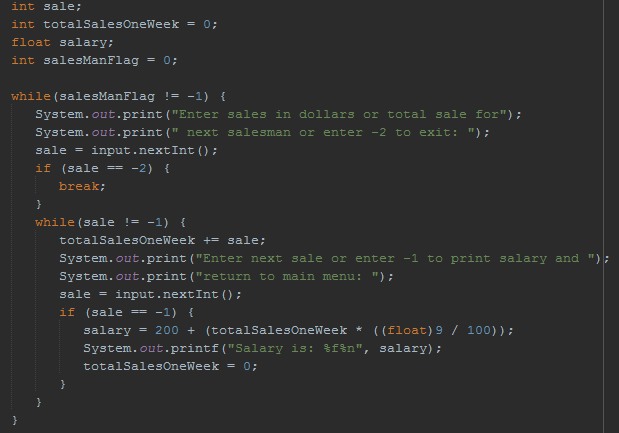
System.out.println("Count is: " + count);

if (count == 5)

continue;

count++;

}



**-for:** Contains elements of counter controlled repetition in one line. The increment happens after body is executed. Then the continuation test is performed again.  
 The first line including the keyword for and the paranthesis of for(with everything inside it) is called for statement header.

-Pre increment or post increment: Doesnt matter. Because for statement is equal to following while statement in Java.

for (ForInit ; Expression ; ForUpdate) // Initialization, continuation test, increment

forLoopBody(); // Body

Initialization is done only once at the beginning as you can see in the same example below.

ForInit; // Initialization

while (Expression) { // Continuation test

forLoopBody(); // Body

ForUpdate; // Increment

}

The increment expression in a for acts as if it were a standalone statement at the end of the for’s body. Therefore, the expressions

counter = counter + 1

counter += 1

++counter  
 counter++

are equivalent increment expressions in a for statement. Many programmers prefer counter++

because it’s concise and because a for loop evaluates its increment expression *after* its body executes, so the postfix increment form seems more natural. In this case, the variable being incremented does not appear in a larger expression, so preincrementing and postincrementing have the *same* effect.

-Using multiple variables in for header: You seperate initializations of variables using commas. But you cant seperate expressions with a commas. You have to merge the simple conditions into a complex condition.

- for (byte column = 0, row = 0; column <= 2 && row >= 0; column++, row--)

- for (byte counter = 39, counter2 = 1; counter > 39 - string1.length();   
 counter--, counter2++)

{

hugeInteger1[index][counter] = Character.getNumericValue(

string1.charAt(string1.length() - counter2));

}

Off-by-oneerror is the error when your loop executes one time more or less then you wanted or one more or less than where you wanted it to start from. To get over this problem generally use 1 and <=, >=. If you have to use 0 then use <, >.

-Sentinel controlled for

for(int sentinelValue = 0; sentinelValue >= 0; ) {

System.out.print("Enter a non negative value or enter a negative value to exit: ");

sentinelValue = input.nextInt();

System.out.println(sentinelValue);

}

-Omitting: All three expressions in for header are optional.

If the loopContinuationCondition is omitted, Java assumes that the loop-continuation condition is always true, thus creating an infinite loop.

You might omit the initialization expression if the program initializes the control variable before the loop.

You might omit the increment expression if the program calculates the increment with statements in the loop’s body or if no increment is needed.

The increment of a for statement may also be *negative*, in which case it’s a decrement, and

the loop counts *downward*.

-Using control variable’s value: Although the value of the control variable can be changed in the body of a for loop, avoid doing so, because this practice can lead to subtle errors. You can assign it to a temporary variable and use that but be careful.

-Using equality operators in continuation condition: Do not use equality operators (!= or ==) in a loop-continuation condition if the loop’s control variable increments or decrements by more than 1. For example, consider the for statement header

for (int counter = 1; counter != 10; counter += 2)

The loop-continuation test counter != 10 never becomes false (resulting in an infinite loop) because counter increments by 2 after each iteration.

-**do while:** The do…while statement tests the loop-continuation condition after executing the loop’s body; therefore, the body always executes at least once.

do {

System.out.println("Count is: " + count);

count++;

} while (count < 11);

-Braces in a do…while Repetition Statement: It isn’t necessary to use braces in the do…while repetition statement if there’s only one statement in the body. However, many programmers include the braces, to avoid confusion between the while and do…while statements.

**-switch:** The switch multiple-selection statement performs different actions based on the possible values of a constant integral expression of type byte, short, int or char. As of Java SE 7, the expression may also be a String.

The switch statement consists of a block that contains a sequence of case labelsand an optional default case.

The break statementcauses program control to proceed with the first statement after the switch.

The switch statement does *not* provide a mechanism for testing *ranges* of values, so *every* value you need to test must be listed in a separate case label. Each case can have multiple statements. The switch statement differs from other control statements in that it does *not* require braces around multiple statements in a case.

-“switch” is faster than “if, else if”. Difference is not big for just few items but “if, else if” executes sequentially meanwhile switch uses a table to find a match.

-Case without a break statement: Without break statements, each time a match occurs in the switch, the statements for that case and subsequent cases execute until a break statement or the end of the switch is encountered. This is often referred to as “falling through” to the statements in subsequent cases. This happens because case is not a control statement. It is a label. Switch statement starts the execution from the matching case and executes until end of switch block(including default case) unless we hit a break.

In the example below the same code executes for case 9 and 10. Because If case is 10, it executes code of 10 and breaks. If case is 9 then is doesnt execute anything for 9 but it falls to case 10 because there is no break and it executes code of case 10. Execution would fall to case 10 even if case 9 had statements inside.

case 9: // grade was between 90

case 10: // and 100, inclusive

++aCount;

break; // exits switch

Cases can be written horizontally too.

switch (getState())

{

case "MA": case "NJ": case "NY": case "PA":

noFaultState = true;

break;

}

-The default case: If no match occurs between the controlling expression’s value and a case label, the default case executes. If no match occurs and the switch does not contain a default case, program control simply continues with the first statement after the switch.

Default case is usually put at the last. And even though it is last usually we we still use break statement to make default case look like others. You can use default case somewhere other than bottom too.

Using default makes your code robust aganist exceptions.

-Expressions in cases of switch: When using the switch statement, remember that each case must contain a constant integral expression—that is, any combination of integer constants that evaluates to a constant integer value (e.g., –7, 0 or 221)—or a String. An integer constant is simply an integer value. In addition, you can use character constants—specific characters in single quotes, such as 'A', '7' or '$'—which represent the integer values of characters and enum constants.

The expression in each case can also be a constant variable—a variable containing a

value which does not change for the entire program. Such a variable is declared with keyword

final. Enum type constants can also be used in case labels.

We can also use constant String expression for switch controlling expression which consist of string literals and constant string variables.

**-Enhanced For:** [enhanced\_for](#enhanced_for)

**-Unconditional control statements**

**-break:** The break statement causes immediate exit from repetition statements or switch statement.

for (count = 1; count <= 10; count++) { // loop 10 times

if (count == 5)

break;

System.out.printf("%d ", count);

}

Some programmers feel that break and continue violate structured programming Since the same effects are achievable with structured programming techniques, these programmers do not use break or continue.. For break statement one way of replacing is adding extra control expression.

for (count = 1; count <= 10 && count != 5; count++)

System.out.printf("%d ", count);

-Label break: Used to exit out of more than one nested loop. The flow of control is not transfered to label. Label just shows which repetition statement we broke from. When we are in nested repetition statements, we can break out of one of these repetition statements using label break. Which means when we are inside inner repetition statement below, writing break outer1 or outer2 works but break outer3 wont compile.

first:

for (int i = 0; i < 10; i++) {

second:

for (int j = 0; j < 5; j++) {

break first;

}

}

third:

for (int a = 0; a < 10; a++)

...

**-continue:** The continue statement skips the remaining statements in the loop body and proceeds with the next iteration of the loop.

In while and do…while statements, the program evaluates the loop-continuation test immediately after the continue statement executes.

In a for statement, the increment expression executes, then the program evaluates the loop-continuation test.

for (int count = 1; count <= 10; count++) { // loop 10 times

if (count == 5)

continue;

System.out.printf("%d ", count);

}

To avoid using continue you can use if, if else statements to avoid executing code for a condition.

for (int count = 1; count <= 10; count++) {

if (count != 5)

System.out.printf("%d ", count);

}

-Label continue:Both continue first and continue second skip to next iteration of outer loop.

first:

for (int i = 0; i < 10; i++) {

second:

for (int j = 0; j < 5; j++) {

if (j == 5)

continue first;

}

}

third:

for (int a = 0; a < 10; a++)

...

**-method call** is fine for structured programming. It takes us to the beginning of the method.

**-return:** Havingmultiplereturn statementspushes limits of structured programming. So if you want to be a structured programming purist, keep your methods small and use one return statement per method.

* **Methods: A Deeper Look**

You write Java programs by combining new methods and classes with predefined ones available in the Java class library and in various other class libraries.

The Java API provides a rich collection of predefined classes that contain methods for performing common mathematical calculations, string manipulations, character manipulations, input/output operations, database operations, networking operations, file processing, error checking and more.

Don’t reinvent the wheel. When possible, reuse Java API classes and methods. This reduces

program development time and avoids introducing programming errors.

-Divide and Conquer with Classes and Methods: Classes and methods help you modularize a program by separating its tasks into self-contained units. Methods are declared within classes. Classes are typically grouped into packages.

The statements in the method bodies are written only once, are hidden from other methods and can be reused from several locations in a program.

One motivation for modularizing a program into classes and methods is the *divide-and-*

*conquer* approach, which makes program development more manageable by constructing

programs from small, simple pieces.

Another is software reusability—using existing classes and methods as building blocks to create new programs.

A third motivation is to *avoid repeating code*. Dividing a program into meaningful classes and methods makes the program easier to debug and maintain.

To promote software reusability, every method should be limited to performing a single,

well-defined task, and the name of the method should express that task effectively.

A method is invoked by a method call, and when the called method completes its task, it returns control and possibly a result to the caller.

The boss method(the method that calls other methods) does not know how the worker method performs its designated tasks. The worker may also call other worker methods, unbeknown to the boss. This “hiding” of implementation details promotes good software engineering.

You should control return values of methods and act accordingly if the method was succesful or not.

Java is pass by value(call by value) (as opposed to pass by reference(call by reference)). We dont pass pointers. If you need to pass a value or an address(reference) you copy the value or the adress and pass that. You dont pass the original pointer.

**-Static:** [static\_part1](#static_part1)

-Why is method main declared static:When you execute the JVM with the java command, the JVM attempts to invoke the main method of the class you specify—at this point no objects of the class have been created. Declaring main as static allows the JVM to invoke main without creating an instance of the class. When you execute your application, you specify its class name as an argument to the java command, as in

java *ClassName argument1 argument2* …

The JVM loads the class specified by *ClassName* and uses that class name to invoke method main. In the preceding command, *ClassName* is a command line argumentto the JVM that tells it which class to execute. Following the *ClassName*, you can also specify a list of Strings (separated by spaces) as command line arguments that the JVM will pass to your application.

**-Final:**

-Final variables. Any field declared with keyword final is *constant*—its value cannot change after the field is initialized.

You can either assign a value to final variables initially or in constructor. Final fields must be initialized in its declaration or in every constructor or a compilation error occurs.

private static final int SNAKE\_EYES = 2;

-Final reference variable. You can’t reassign another object to that reference variable. But you can use methods and variables, and change variables of this object.

final A ob = new A();

ob.setI(6) // Works

final A ob = new A();

ob = new A(); // Doesn’t work

-Final methods are inherited but can't be overridden. This allows the original developer to create functionality that cannot be changed by subclasses. Calls to final methods are resolved at compile time—this is known as static binding. [dynamic\_binding](#dynamic_binding)

-Fınal class. A final classcannot be extended to create a subclass (final classes cannot be superclasses). All methods in a final class are implicitly final. Class String is an example of a final class. If you were allowed to create a subclass of String, objects of that subclass could be used wherever Strings are expected. Since class String cannot be extended, programs that use Strings can rely on the functionality of String objects as specified in the Java API. Making the class final also prevents programmers from creating subclasses that might bypass security restrictions.

Unless you carefully design a class for extension, you should declare the class as final to avoid (often subtle) errors.

Methods can return at most one value, but the returned value could be a reference to an object that contains many values.

Variables should be declared as fields only if they’re required for use in more than one method

of the class or if the program should save their values between calls to the class’s methods.

You can use a methods call(the value returned bu that method call) as and argument to another method. This way you can implement software reuse.

return Math.max(x, Math.max(y, z));

**-Declaring and using methods**

**1-** Using a method name by itself to call another method of the *same* class—such as

maximum(number1, number2, number3) // pg 247-248

**2-** Using a reference to an object, followed by a dot (.) and the method name to call a non-static method of the referenced object—such as

myAccount.getName // pg 116

**3-** Using the class name and a dot (.) to call a static method of a class—such as

Math.sqrt(900.0) // pg 245

**-Returning a method**

**1-** Reaching right brace(closing brace).

**2-** return;

**3-** return value;

A static method can call other static methods of the same class directly (i.e., using the method name by itself) and can manipulate static variables in the same class directly. To access the class’s instance variables and instance methods, a static method must use a reference to an object of the class. Instance methods can access all fields (static variables and instance variables of it’s own object).

Recall that static methods relate to a class as a whole, whereas instance methods are associated with a specific instance (object) of the class and may manipulate the instance variables of that object.

**-Accessing Methods and Fields**

Instancemeans non static (belongs to instance/object).

Access modifiers are important when determining scope.

In order to use instance variables/methods you need to create an instance first.

Use “this” for instance variable/methods, use “ClassName.\_\_\_” For static variables/methods.

**-Same Class**

**-Static variable:** We can access a static variable from instance and static methods directly if there is no shadowing local variable. If there is, access it with ClassName.variableName.

**-Static method:** We can access a static method from instance and static methods directly.

**-Instance variable:** They are generally private. So they are generally used in methods of their own class. We can access instance variables from instance methods directly if there is no shadowing. If there is, access it with “this.variable.Name”. We can’t access instance variables directly from static methods. From static methods, after making sure that object is defined in your static method(either declare in static method or send object’s reference as argument) use objectName.variableName;

Objects of same class can access each others private instance variables but its not recommended. Use get methods instead.

**-Instance method:** We can access instance methods directly from instance methods. We cant access instance methods directly from static methods. From static methods, after making sure that object is defined in your static method(either declare in static method or send object’s reference as argument) use objectName.methodName;

**-Another Class (imported):** Different objects have to have different names so you dont have to worry about object or method name collision.

**-Static variable:** If we are in its scope (determined by its access modifier), we can access it from instance and static methods with ClassName.variableName.

**-Static method:** If we are in its scope (determined by its access modifier), we can access it from instance and static methods with ClassName.methodName().

**-Instance variable:** They are generally private. So they are generally used in methods of their own class. If its not private and we are in its scope (determined by its access modifier), after making sure that object is defined in your instance or static method(either declare in method or send object’s reference as argument) access it using objectName.variableName;

**-Instance method:** If we are in its scope (determined by its access modifier), after making sure that object is defined in your your instance or static method(either declare in method or send object’s reference as argument) access it from instance and static methods using objectName.methodName().

**-Method Call Stack and Stack Frames:** When a program calls a method, the called method must know how to return to its caller, so the return address of the calling method is pushed onto the method-call stack. If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out(first-in, last out) order so that each method can return to its caller.

The method-call stack also contains the memory for the local variables (including the method parameters) used in each invocation of a method during a program’s execution. This data, stored as a portion of the method-call stack, is known as the stack frame (or activation record) of the method call. When a method call is made, the stack frame for that method call is pushed onto the method-call stack. When the method returns to its caller, the stack frame for this method call is popped off the stack and those local variables are no longer known to the program. If a local variable holding a reference to an object is the only variable in the program with a reference to that object, then, when the stack frame containing that local variable is popped off the stack, the object can no longer be accessed by the program and will eventually be deleted from memory by the JVM during garbage collection.

Of course, a computer’s memory is finite, so only a certain amount can be used to store stack frames on the method-call stack. If more method calls occur than can have their stack frames stored, an error known as a stack overflow occurs.

Stack is used for static memory allocation and Heap for dynamic memory allocation, both stored in the computer’s RAM. [Stack and Heap](http://net-informations.com/faq/net/stack-heap.htm)

**-Argument Promotion and Casting:** [implicit\_conversion](#implicit_conversion)Converting an argument’s value, if possible, to the type that the method expects to receive in its corresponding parameter.

Conversions may lead to compilation errors if Java’s promotion rules are not satisfied. These rules specify which conversions are allowed—that is, which ones can be performed without losing data.

The promotion rules apply to expressions containing values of two or more primitive types and to primitive type values passed as arguments to methods. Each value is promoted to the “highest” type in the expression. Actually, the expression uses a temporary copy of each value—the types of the original values remain unchanged.

In cases where information may be lost due to conversion, the Java compiler requires you to use a cast operator to explicitly force the conversion to occur.

square((int) doubleValue)



Long can be assigned to float because floating point variables are held in a different way than integers [Binary representation.docx](file:///C:\Computer%20Engineering\8-%20Java\3-%20Notes\-%20Other\Binary%20representation.docx). Which allow 32 bit float to hold value of 64 bit long.

**-Secure Random-Number Generation** (java.security.SecureRandom, java.util.Random):The element of chance can be introduced in a program via an object of class Secure-Random. Such objects can produce random boolean, byte, float, double, int, long and Gaussian values. [Gaussian distribution](http://www.javamex.com/tutorials/random_numbers/gaussian_distribution_2.shtml) [Generating a random character](https://stackoverflow.com/questions/2626835/is-there-functionality-to-generate-a-random-character-in-java)

Recent editions of this book used Java’s Random class to obtain “random” values. This class

produced deterministic values that could be predicted by malicious programmers. Secure-Random objects produce nondeterministic random numbers that cannot be predicted. Nondeterministic algorithms can exhibit different behaviour even for same input.

[Lineral congruential generator](https://en.wikipedia.org/wiki/Linear_congruential_generator) [Cryptographically secure pseudorandomnumbergenerator](https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator)

SecureRandom randomNumbers = new SecureRandom();

int randomValue = randomNumbers.nextInt();

If it truly produces values *at random*, then every value in the range should have an *equal chance* (or probability) of being chosen each time nextInt is called.

SecureRandom provides another version of method nextInt that receives an int argument and returns a value from 0 up to, but not including, the argument’s value.

int face = randomNumbers.nextInt(6);

The argument 6—called the scaling factor—represents the number of unique values that nextInt should produce (in this case six—0, 1, 2, 3, 4 and 5). This manipulation is called scaling the range of values produced by SecureRandom method nextInt.

A six-sided die has the numbers 1–6 on its faces, not 0–5. So we shiftthe range of numbers produced by adding a shifting value—in this case 1—to our previous result, as in

int face = 1 + randomNumbers.nextInt(6);

int number = *shiftingValue* + randomNumbers.nextInt(*scalingFactor*);

where shiftingValue specifies the first number in the desired range of consecutive integers and scalingFactor specifies how many numbers are in the range.

It’s also possible to choose integers at random from sets of values other than ranges of

consecutive integers. For example, to obtain a random value from the sequence 2, 5, 8, 11

and 14, you could use the statement

int number = 2 + 3 \* randomNumbers.nextInt(5);

int number = *shiftingValue* + *differenceBetweenValues* \* randomNumbers.nextInt(*scalingFactor*);

where *shiftingValue* specifies the first number in the desired range of values, *difference-BetweenValues* represents the *constant difference* between consecutive numbers in the sequence and *scalingFactor* specifies how many numbers are in the range.

1, 2, 4, 8, 16, 32 -> n = Math.pow(2, randomNumbers.nextInt(6));

-A Note About Performance:Using SecureRandom instead of Random to achieve higher levels of security incurs a significant performance penalty. For “casual” applications, you might want to use class Random.

**-enum**:[enum part2](#enum_part2) Enum declares a set of identifiers which are constants. These constants are called enum constants. So we use constant identifier conventions while naming them. But these are not variables. They are the values.

Defining new enum types cant be done in methods. Must be done in field.

private enum Status { CONTINUE, WON, LOST };

Status gameStatus;

You cant assign strings to an enum type variable. You have to assign one of its values to it.

- gameStatus = “Won”; // Wrong

gameStatus = Status.WON; // Correct

- while (gameStatus == Status.CONTINUE)

-Why do we use enum and final primitive variablesUsing enum constants instead of literals like 0, 1, 2 make the program easier to read and maintain. But using an enum becomes unpractical if you need to compare the enum value with something else or use it in a calculation or an expression etc. You can do it with switch statements but it defeats the purpose of using enums which is making the program more readable. So in these places use final variables. Named constants can be declared using both enum types and final variables (Named constants is a general name for these two). But enum type you define in field is known to all instances of class and class itself so enums are implicitly static finals.

**-Method Overloading:** Methods of the same name can be declared in the same class, as long as they have different sets of parameters (determined by the number, types and order of the parameters)—this is called method overloading. When an overloaded method is called, the compiler selects the appropriate method by examining the number, types and order of the arguments in the call.

obj.sum(10);

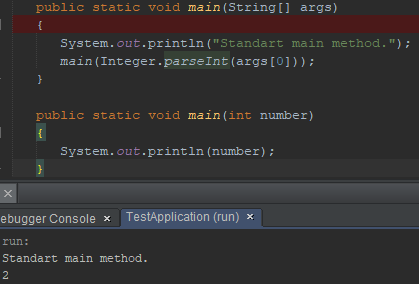
obj.sum(20, 20);

-Distinguishing Between Overloaded Methods:The compiler distinguishes overloaded methods by their signatures—a combination of the method’s name and the number, types and order of its parameters, but not its return type.

Internally, the compiler uses longer method names that include the original method name, the types of each parameter and the exact order of the parameters to determine whether the methods in a class are unique in that class.

-Return Types of Overloaded Method:Overloaded methods can have different return types if the methods have different parameter lists. Method calls cannot be distinguished only by return type. Return types of methods alone doesnt tell us which method we should use because we cant know what the return type is before executing the method with the input.

-Overloading main method:You can overload main method in Java. But the program doesn't execute the overloaded main method when you run your program, you have to call the overloaded main method from the original main method.



-varargs in method overloading:You can’t use varargs in overloading.

**-Variable amount of parameters(varargs):** You can only have one varArg per method. Must be at the end of parameter list.  
  
 public static void test(String... stringArray)  
 {  
 int x = stringArray.length;  
 System.out.println(stringArray[1]);  
 String a = stringArray[1];  
 }  
  
 public static void main(String[] args)  
 {  
 test("ab", "cd", "he", "ke");  
 }

-Difference between array parameter and varargs parameter: If we were to use array instead of varargs as stringArray parameter in test method, it would not compile. It would require   
  
 test(new String[] { "ab", "cd", "he", "ke" });

* **Arrays and ArrayLists**

Array objects are data structures consisting of related data items of the same type. Arrays remain same length once they are created.

Variables in an array are called elements or components.

Arrays are objects in Java. A reference variable in stack refers to an array object in heap.

Elements can be primitive or reference types (including arrays).

Every element of a primitive-type array contains a variable of the array’s declared type. Every element of a reference-type array is a reference to an object of the array’s declared type.

**-Declaring and initializing arrays:** To create an array object, you specify the type of the array elements and the number of elements as part of an array creation expression that uses keyword new. Such an expression returns a reference that can be stored in an array variable.

Their size must be declared before using them. Every array object knows its own length and stores it in a length instance variable. Even though the length instance variable of an array is public, it cannot be changed because it’s a final variable. There are two ways of declaring its size as shown below.

Elements of arrays get default values of their data type when the array is created. (Both locals and fields.)

- int[] array;

array = new int[10]; // Declaration of array reference and creation of array done discretely.

- int[] array = new int[10]; /\* Declaration of array reference and creation of array done together. Declares an int array reference variable, then initializes the variable with a reference to an array object containing 10 int elements. \*/

- int[] array1, array2;

array1 = new int[] {1, 10, -7};  
 array2 = new int[] {no1, no2, no3}; // You can use variables too.

Notice how you need a cast for the option above but not the option below.

- int[] array = {1, 10, -7}; /\* Array initializer(complete statement) and the intializer list. Length of the array is determined by the number of elements in the initializer list. Compiler actually takes a look at your array initializer and sets up the appropriate array declaration and initialization using new behind the scenes. \*/

Creating an array in an argument list. Then turning the array to a string.

public class Foo {

public static void method(String[] myStrArray) {

System.out.println(Arrays.toString((myStrArray)));

}

public static void main(String[] args) {

method(new String[]{"hello", "goodbye"}); // \*\*array created inline\*\*

}

}

Multiple arrays can be created in a single declaration. But its not recommended and it can cause subtle errors.

To refer to a particular element in an array, we specify the name of the reference to the array and the position number of the element in the array (the element’s index or subscript) in square brackets ( [] ). This is called an array access expression. First index is 0. (Zero based counting.) Indexes can be any non negative integer value other then long. Expressions can be used as indexes.

myArray[0] = 10;

myArray[a + b] = 5;

When only one variable is declared in each declaration, the square brackets can be placed either after the type or after the array variable name. But placing the square brackets after the type is preferred. Same for method parameters.

Arrays of objects.

private Card[] deck; /\* Declaration of array (reference variable that shows the   
 array object on heap). Part 1 of array creation expression. \*/

private static final int NUMBER\_OF\_CARDS = 52;

public DeckOfCards() {

deck = new Card[NUMBER\_OF\_CARDS]; /\* Creation of the array with 52 elements

(reference variables that reference to card objects) that have default   
 values (null). Part 2 of array creation expression. \*/

for (int count = 0; count < deck.length; count++)

deck[count] = new Card(faces[count % 13], suits[count / 13]);   
 /\* Initialization of array elements (creating Card objects and assigning   
 addresses of these objects to reference variables in array). \*/

}

**-ArrayIndexOutOfBounds Exception:** [exception handling part2](#exception_handling_part2) An exception indicates a problem that occurs while a program executes. Exception handling helps you create fault-tolerant programs that can resolve (or handle) exceptions. When the JVM or a method detects a problem, such as an invalid array index or an invalid method argument, it throws an exception—that is, an exception occurs.

The try block contains the code that might throw an exception, and the catch block contains the code that handles the exception if one occurs. You can have many catch blocks to handle different types of exceptions that might be thrown in the corresponding try block.

When an exception is thrown the try block is terminated and catch block starts executing. If you declared any local variables in the try block, they are now out of scope. The catch block declares an exception parameter (e) of type (ArrayIndexOutOfBoundsException). The catch block can handle exceptions of the specified type. Inside the catch block, you can use the parameter’s identifier to interact with a caught exception object.

-How to handle it:When writing code to access an array element, ensure that the array index remains greater than or equal to 0 and less than the length of the array. This would prevent “ArrayIndexOutOfBoundsException”s.

-Implicit toString and get/set methods:When the code in catch block executes,

catch (ArrayIndexOutOfBoundsException e) {

System.out.println(e); // invokes toString method

the toString method of exception object is implicitly called to get the error message that’s stored in the exception object and display it.

When you use a non string variable where a string is expected, its class’s toString method executes implicity. For this effect to occur, toString must be declared with the header below

public String toString()

**-NullPointerException:** Occurs when you try to call a method on a null reference. In industrial-strength code, ensuring that references are not null before you use them to call methods prevents NullPointerExceptions.

**-Enhanced For:** The enhanced for statement iterates through the elements of an array or a collection without using a counter, thus avoiding the possibility of “stepping outside” the array or other errors due to the initial value of counter, loop continuation test and increment expression. But if you need to use counter in your program, you need to use the counter controlled for statement.

The type of the parameter must be consistent with the type of the elements in the array.

The statement’s header can be read as “for each iteration, assign the next element of array to int variable number, then execute the following statement.”

The enhanced for statement can be used only to obtain array elements—it cannot be used to modify elements. Because iteration variable is just a copy of the array element.

Using enhanced for with one dimensional arrays:

for (int counter = 0; counter < array.length; counter++)  
 total += array[counter];  
  
 for (int number : array)  
 total += number;  
  
When naming the “counter” in enhanced for, dont use names that would describe counter. Use names that would describe array[counter] such as “value” instead of “counter”.  
  
Using enhanced for with multidimensional arrays: As long as you have more dimensions below the current array, the left value is an array. Thats why we are able to use that left value(arrayRow) in the place of where an array is supposed to go in the statement below. But the left value below(arrayColumn) is not an array. You can see which variable is what data type and how many dimensions it has by looking at its data type left of it. arrayRow is an int[] while arrayColumn is an int.  
  
 for (int[] arrayRow : array)  
 for (int arrayColumn : arrayRow)  
 System.out.println(arrayColumn);  
  
**-Passing arrays to methods:** When an argument to a method is an array or a reference type array element, the called method receives a copy of the reference. However, when an argument to a method is a primitive type array element, the called method receives a copy of the element’s value. Such primitive values are called scalars or scalar quantities. [Scalar vs primitive data type](https://stackoverflow.com/questions/6623130/scalar-vs-primitive-data-type-are-they-the-same-thing)

You can read and modify elements of an array if you pass array’s reference (or reference type array element) to the method. But you cant modify element of an array if you pass a primitive type element.

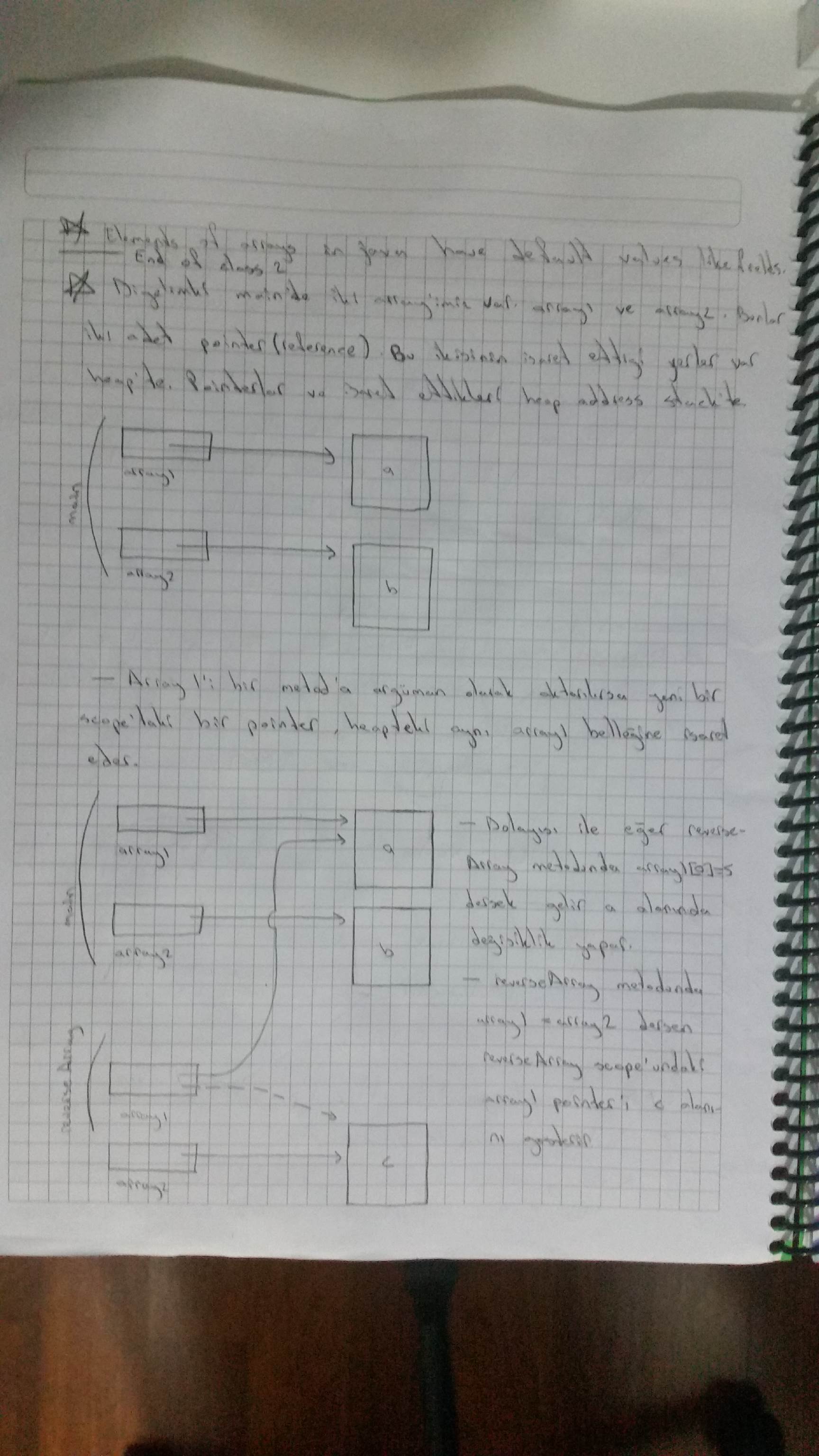
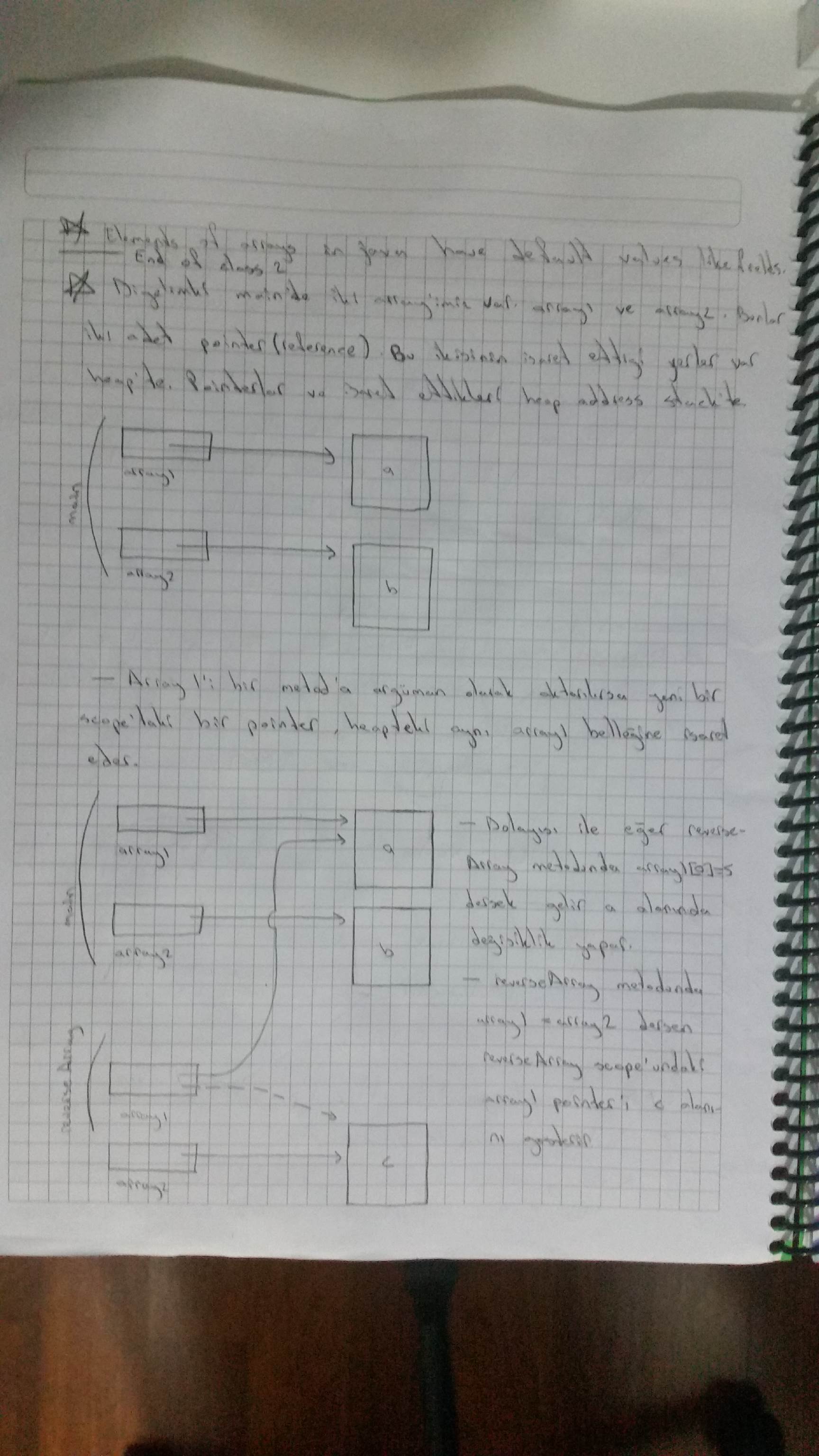
calculateSqrt(array[3]); // Copying one element’s value to the parameter in method.

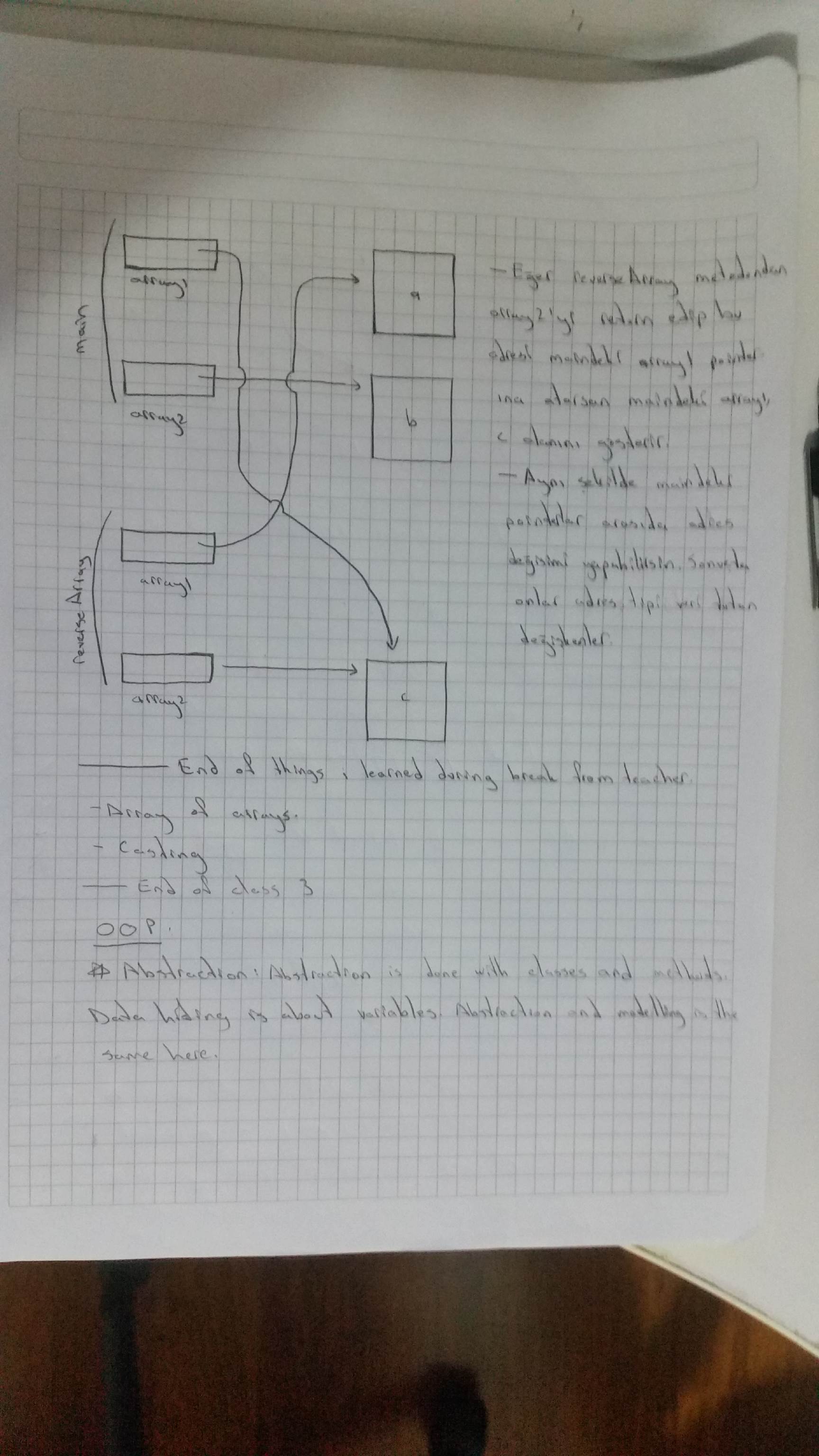
modifyArray(array); // Copying the reference of the array to the parameter in method.  
  
**-Pass by Reference, Pass by Value:** There are two ways of passing arguments to methods.  
  
-In other languages: When an argument is passed by value, a copy of the argument’s value is passed to the called method. The called method works exclusively with the copy. (Which is in stack) Changes to the called method’s copy do not affect the original variable’s value in the caller.  
  
When an argument is passed by reference, the called method can access the argument’s value in the caller directly and modify that data. Pass-by-reference improves performance by eliminating the need to copy possibly large amounts of data.  
  
-In Java: Unlike some other languages, Java does not allow you to choose pass-by-value or pass by-reference—all arguments are passed by value. A method call can pass two types of values to a method—copies of primitive values (e.g., values of int and double) and copies of references to objects.

When a method modifies a primitive-type parameter, changes to the parameter have no effect on the original argument value in the calling method. When it comes to objects, objects themselves cannot be passed to methods so we pass address of the object held in reference variable.

Inside a method, if you modify a reference-type parameter so that it refers to another object, only the parameter refers to the new object—the reference stored in the caller’s variable (argument) still refers to the original object.

Although an object’s reference is passed by value, a method can still interact with the referenced object by calling its public methods using the copy of the object’s reference. Since the reference stored in the parameter is a copy of the reference that was passed as an argument, the parameter in the called method and the argument in the calling method refer to the sameobject in memory.

Passing references to arrays, instead of the array objects themselves, makes sense for performance reasons. Because everything in Java is passed by value, if array objects were passed, a copy of each element would be passed. For large arrays, this would waste time and consume considerable storage for the copies of the elements.  
  
  


  
**-Multi-Dimensional arrays:** Java does not support multidimensional arrays directly, but it allows you to specify one-dimensional arrays whose elements are also one-dimensional arrays(arrays of arrays, arrays of one dimensional arrays), thus achieving the same effect. You can nest this to get as many dimensions as you want. This also allows variable length multidimensional arrays.

You have to specify the length of the first dimension. This is basically the same thing as an “array must have a length”. You havent created the lower level arrays yet. So you can specify their length later when you create them.

An array with m rows and n columns is called an m-by-n array.

int[][] array = new int[2][3]; // Array creation expression.

int[][] array = {{ 1, 2, 3 }, { 4, 5, 6 }}; /\* Nested array initializer. Compiler counts number of nested array initializers to determine the number of rows in array named “array”. The compiler counts the initializer values in the nested array initializer for a row to determine the number of columns in that row. Every row can have different length. \*/

array1[0][0] = 10; // Example to value assignment.  
  
-Multi-Dimensional arrays with different lengths:All arrays have to be the same data type.

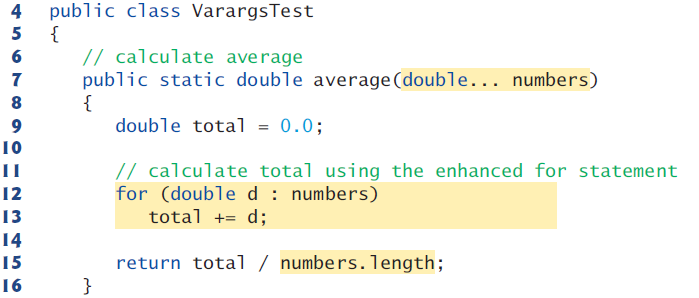
int[][] array = {{ 1, 2 }, { 3, 4, 5 }}; // First way.

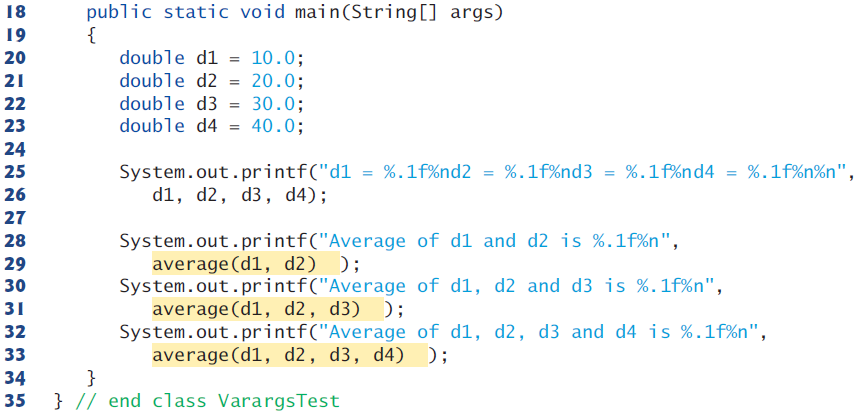
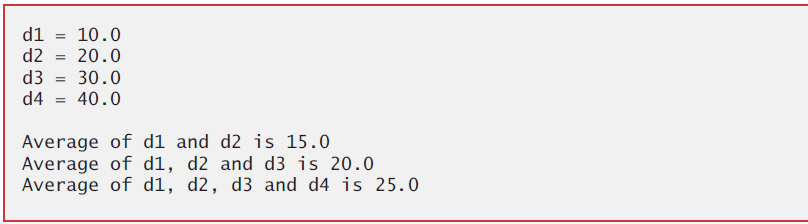
int[][] array = new int[2][]; /\* Second way. Length of the array is 5. Holds 5 reference for 5 one dimensional arrays. Array creation expression can use literal values or variables to specify array dimensions, because new creates arrays at execution time—not at compile time. \*/

array[0] = new int[2]; // When we use this second way we have to use new.

array[1] = new int[5];

-Traversing multi dimensional arrays:   
  
 for (int row = 0; row < array.length; row++) // Traversing with for  
 {  
 for (int column = 0; column < array[row].length; column++)  
 System.out.printf("%d ", array[row][column]);  
 System.out.println();  
 }  
  
 for (int[] arrayRow : array) // Traversing with enhanced for  
 {  
 for (int arrayColumn : arrayRow)  
 System.out.printf("%d ", arrayColumn);  
 System.out.println();  
 }

-Traversing a certain row(or lower dimension) of a multi dimensional array:  
  
 for (int column = 0; column < array[1].length; column++) // Traversing with for  
 System.out.printf("%d ", array[1][column]);  
  
 for (int column : array[1]) // Traversing with enhanced for  
 System.out.printf("%d ", column);  
  
**-Variable Length Argument Lists:** With variable-length argument lists, you can create methods that receive an unspecified number of arguments. A type followed by an ellipsis (...) in a method’s parameter list indicates that the method receives a variable number of arguments of that particular type. This use of the ellipsis can occur only once in a parameter list, and the ellipsis, together with its type and the parameter name, must be placed at the end of the parameter list.  
 It has to be the only varargs so the compiler knows which argument is for which parameter. Otherwise the compiler wouldnt know if an argument belongs to varargs or the parameter after varargs.  
 The method body access variable number of parameters as an array of that data type.  
 You can supply an array as an argument for this parameter.  
  


  
  
  
 **-Command Line Arguments:** It’s possible to pass arguments from the command line to an application via method main’s String[] parameter, which receives an array of Strings. By convention, this parameter is named args. When an application is executed using the java command, Java passes the command line arguments that appear after the class name in the java command to the application’s main method as Strings in the array args. When we execute an application we call class’s main method with arguments (if no arguments are entered then the string is empty) (tag: main method argument)  
 Common uses of command line arguments include passing options and filenames to applications.  
 Command line arguments are separated by white space, not commas. If the data you want to enter includes spaces than use quotes around your string.  
  
 java initArray 50 0 40 /\* This is how we executed the program from command   
 line. So we have an args string array with 3 strings. \*/

public static void main(String args[])

{

int exam1 = Integer.parseInt(args[0]);

int exam2 = Integer.parseInt(args[1]);

int exam3 = Integer.parseInt(args[2]);

}

**-Array methods:** For the list of methods and more information on them use IDE autocomplete feature after writing “arrayName.” or API Spec.

- arrayName.length;

- arrayName.equals(arrayName2); // Only checks if pointers are pointing to the same array object.

public boolean equals(Object obj) {

return (this == obj);

}

**-Class Arrays** (java.util.Arrays):[Class Arrays](http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html)

- boolean = Arrays.equals(array1, array2); /\* Checks if pointers are pointing to the same array object, if at least of them is pointing to null, if their length is same. If their elements are same. \*/  
  
 public static boolean equals(int[] a1, int[] a2) {  
 if (a1==a2)  
 return true;  
 if (a1==null || a2==null)  
 return false;  
 if (a2.length != a1.length)  
 return false;  
  
 for (int i = 0; i < a1.length; i++)  
 if (a1[i] != a2[i])  
 return false;  
 return true;  
 }

- Arrays.sort(array);

- Arrays.fill(array, 7); /\* Arrays.fill doesnt work with multiple dimension arrays. You need to divide them into single dimension arrays first. \*/

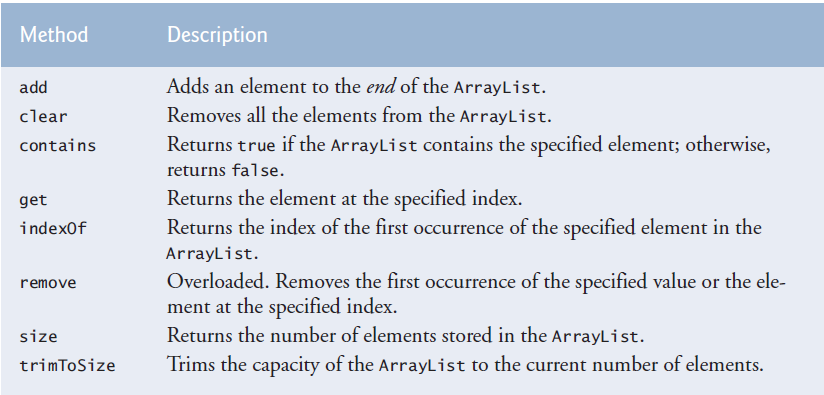
for (int counter = 0; counter < grades.length; counter++)  
 Arrays.fill(grades[counter], -1);

- System.arrayCopy(array, 0, arrayCopy, 0, length); // The first zero is the starting index of the first array which we copy and the second zero is the index where we start copying to.

- Arrays.copyOf(array, length);

int a7[] = Arrays.copyOf(a6, 3);

**-** int location =Arrays.binarySearch(array, 5); /\* If it can’t find the 5 character, it returns a negative value. It can search different data types. Some search algorithms like binary search require sorted arrays. \*/

- Arrays.toString(a8);  
 **-Introduction to Collections and Class ArrayList** (java.util.ArrayList):The Java API provides several predefined data structures, called collections, used to store groups of related objects in memory. These classes provide efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored. This reduces application-development time.   
  
Insert collections\_part2 hyperlink here later. collections\_part1 bookmark has already been created at the header above. Insert collections\_part1hyper link to the collections\_part2 section.  
  
It can dynamically change its size to accommodate more elements.  
  
Declaring list as an ArrayList collection that can store only Strings.  
  
 ArrayList<String> list;  
  
Classes with this kind of placeholder that can be used with any type are called generic classes. Only nonprimitive types can be used to declare variables and create objects of generic classes. However, Java provides a mechanism—known as boxing—that allows primitive values to be wrapped as objects for use with generic classes.  
  
Declaring integers as an ArrayList that can store only Integers. When you place an int value into an ArrayList<Integer>, the int value is boxed (wrapped) as an Integer object, and when you get an Integer object from an ArrayList<Integer>, then assign the object to an int variable, the int value inside the object is unboxed (unwrapped). (tag: unboxing)  
  
 ArrayList<Integer> integers;  
  


Use IDE autocomplete feature or API Spec. to see other methods.

Default initial capacity is 10 elements. The capacity indicates how many items the ArrayList can hold without growing. ArrayList is implemented using a conventional array behind the scenes. When the ArrayList grows, it must create a larger internal array and copy each element to the new array. This is a time-consuming operation. It would be inefficient for the ArrayList to grow each time an element is added. Instead, it grows only when an element is added and the number of elements is equal to the capacity—i.e., there’s no space for the new element.

Inserting an element is usually slower than adding an element to the end of the ArrayList.

Contains method can be inefficient too because it compares its argument to each element of the ArrayList in order.

ArrayList<String> items = new ArrayList<String>();

items.add("red"); // append an item to the list

items.add(0, "yellow"); // insert "yellow" at index 0

items.remove("yellow"); // remove the first "yellow"

items.remove(1); // remove item at index 1

// check if a value is in the List

System.out.printf("\"red\" is %sin the list%n", items.contains("red") ? "": "not ");

System.out.printf("Size: %s%n", items.size()); // display number of elements in the List

for (int i = 0; i < items.size(); i++)

System.out.printf(" %s", items.get(i));  
  
 for (String item : items)  
 System.out.printf(" %s", item);

*-Java SE 7—Diamond (<>) Notation for Creating an Object of a Generic Class*

ArrayList<String> items = new ArrayList<String>();

Notice that ArrayList<String> appears in the variable declaration and in the class instance

creation expression. Java SE 7 introduced the diamond (<>) notation to simplify statements

like this. Using <> in a class instance creation expression for an object of a generic class tells

the compiler to determine what belongs in the angle brackets. In Java SE 7 and higher, the

preceding statement can be written as:

ArrayList<String> items = new ArrayList<>();

When the compiler encounters the diamond (<>) in the class instance creation expression,

it uses the declaration of variable items to determine the ArrayList’s element type

(String)—this is known as inferring the element type.

* **Classes and Objects: A Deeper Look**

**-throw statement:** (Fig. 8.1) The class instance creation expression in the throw statement creates a new object of type IllegalArgumentException. The parentheses following the class name indicate a call to the IllegalArgumentException constructor. In this case, we call the constructor that allows us to specify a custom error message. After the exception object is created, the throw statement immediately terminates method setTime and the exception is returned to the calling method that attempted to set the time.

if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 || second < 0 || second >= 60)  
 throw new IllegalArgumentException("hour, minute and/or second was out of range");

[exception handling part1](#exception_handling_part1) [exception handling part3](#exception_handling_part3)

(Fig 8.2) Statements that can cause exceptions are placed in a try block, in case setTime throws an IllegalArgumentException, which it will do since the arguments are all invalid. When this occurs, the exception is caught at lines 25–28, and line 27 displays the exception’s error message by calling its getMessage method.

try {

time.setTime(99, 99, 99); // all values out of range

}

catch (IllegalArgumentException e) {

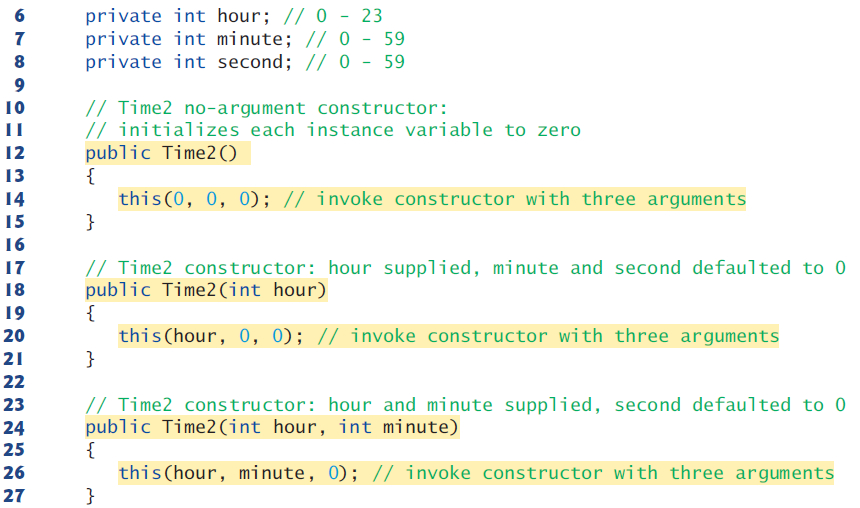
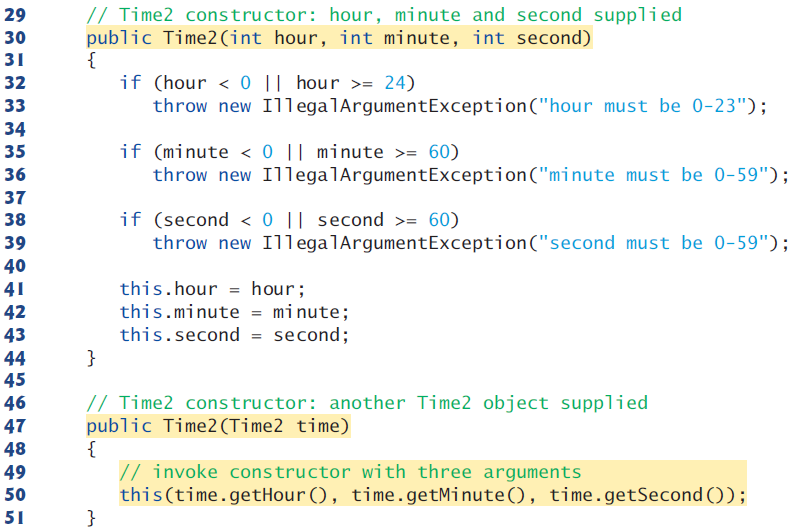
System.out.printf("Exception: %s%n%n", e.getMessage());

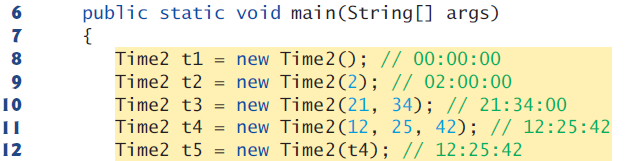
}

-Loose coupling: Classes simplify programming, because the client can use only a class’s public methods. Such methods are usually client oriented rather than implementation. Clients generally care about what the class does but not how the class does it. [coupling\_part2](#coupling_part2)

**-Single method in memory:** (tag: save space)Java conserves storage by maintaining only one copy of each instance method per class (static methods already have one copy)—this method is invoked by every object of the class. Each object, on the other hand, has its own copy of the class’s instance variables. Each method of the class implicitly uses this reference to determine the specific object of the class to manipulate.

**-Overloading constructors:** Constructor calling “this(arguments) or super(arguments)” must be the first statement in constructors or not exist at all.



When one object of a class has reference to another object of the same class, the first object can access all the second object’s data and methods including those that are private (We could have used time.hour, time.minute, time.second in the last constructor). But this is not recommended. When you use get and set methods you just need to maintain and change those. You dont have to maintain and change methods like toUniversalTime or toString when you use get methods in them.

**-Composition:** A class can have references to objects of other classes as members. This is called compositionand is sometimes referred to as a *has-a* relationship. For example an employee object may need to know birth and hire dates. And they can be held as date objects.

public class Employee

{

private String firstName;

private String lastName;

private Date birthDate;

private Date hireDate;

**-enum Types** (java.util.EnumSet):[enum part1](#enum_part1) An enum type is declared with an enum declaration, which is a comma seperated list of enum constants. The declaration may optionally include other components of traditional classes, such as constructors, fields and methods.

-Each enum declaration declares an enum class with the following restrictions:

1. enum constants are implicitly final.

2. enum constants are implicitly static.

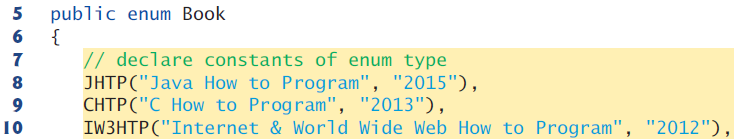
3. Any attempt to create an object of an enum type with operator new results in a

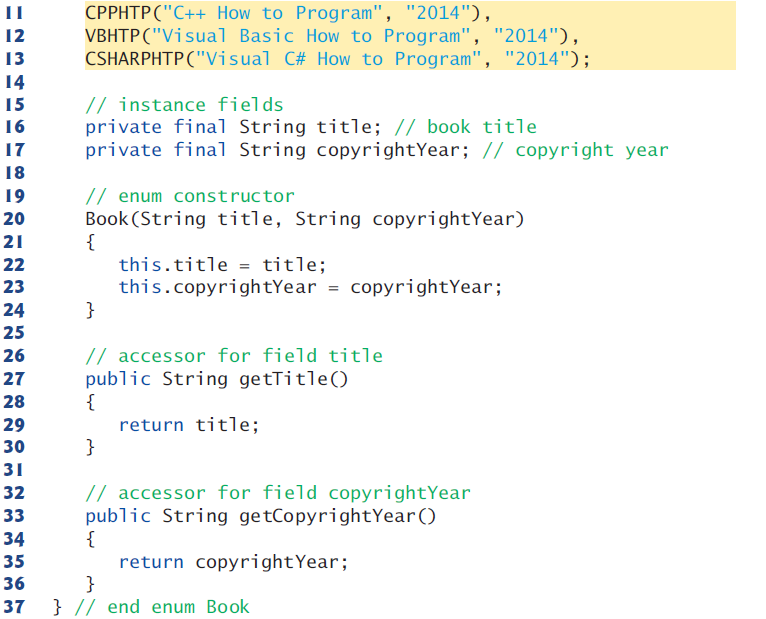
compilation error.

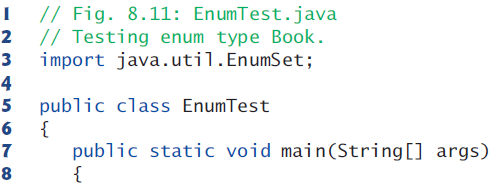
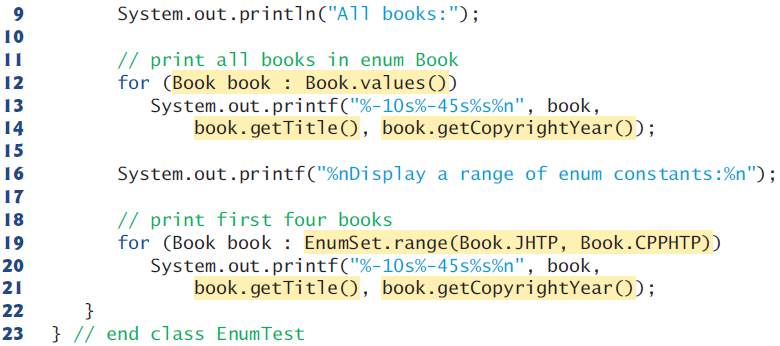
-Declaring Instance Variables, a Constructor and Methods in an enum Type:Constructors of enums are called enum constructors.

For every enum, the compiler generates the static method values that returns an array of the enum’s constants in the order they were declared.

One of the methods of EnumSet is range. Which returns an EnumSet that contains all the constants between these two constants.







We have a constructor but we dont create an object of the enum. The defined constants, gets set with the constructor. Using values and EnumSet.range methods we can act like our enum type is an array of constants.

**-static import:** A static import declaration enables you to import the static members of a class

or interface so you can access them via their unqualified names (as opposed to fully qualified name [package](#package)) in your class—that is, the class name and a dot (.) are not required when using an imported static member.

-Static import forms:A static import declaration has two forms—one that imports a particular static member (known as single static import) and one that imports all static members of a class (known as static import on demand).

import static java.lang.Math.sqrt; // import static *packageN*.*ClassN*.*staticMemberN*;

System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));

import static java.lang.Math.\*; // import static *packageName.ClassName*.\*;

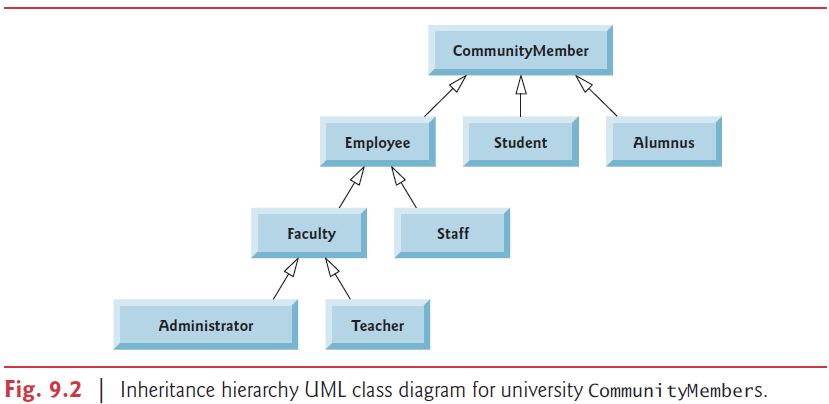
System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));

System.out.printf("ceil(-9.8) = %.1f%n", ceil(-9.8));

System.out.printf("E = %f%n", E);

System.out.printf("PI = %f%n", PI);

* **Object Oriented Programming: Inheritance**

Inheritance is creating a new class by acquiring an existing class’s members and possibly embellishing them with new or modified capabilities. The existing class is called the superclass, and the new class is the subclass. A subclass can become a superclass for future subclasses.   
 A subclass can add its own fields and methods. Therefore, a subclass is more specific than its superclass and represents a more specialized group of objects. The subclass exhibits the behaviors of its superclass and can modify those behaviors so that they operate appropriately for the subclass. This is why inheritance is sometimes referred to as specialization.   
 The direct superclass is the superclass from which the subclass explicitly inherits. An indirect superclass is any class above the direct superclass in the class hierarchy, which defines the inheritance relationships among classes. In Java, the class hierarchy begins with class Object (java.lang), which every class in Java directly or indirectly extends.   
 Java supports only single inheritance, in which each class is derived from exactly one direct superclass. Java uses interfaces to take advantage of multiple inheritance while avoiding the associated problems.  
 We distinguish between the is-a relationship and the has-a relationship. Is-a represents inheritance. In an is-a relationship, an object of a subclass can also be treated as an object of its superclass(direct and indirect)—e.g., a car is a vehicle. By contrast, has-a represents composition. In a has-a relationship, an object contains as members references to other objects— e.g., a car has a steering wheel (and a car object has a reference to a steering-wheel object). (tag: is a relationship, has a relationship)  
  
**-Superclasses and Subclasses**Inheritance relationships form treelike hierarchial structures. Class hierarchy is also called an inheritance hierarchy. Each arrow in the hierarchy represents an is-a relationship. The class at the top is inherited from Object class. So we can say Employee is a CommunityMember and CommunityMember is an Object.  
  
  
  
Objects of all classes that extend a common upperclass can be treated as object of that superclass. This is why we can catch objects of all Exception subclasses with an Exception reference variable. Reference variable of Exception can also refer to objects of subclasses of Exception.  
  
**-Members in inheritance**Public and protected superclass members retain their original access modifier when they become members of the subclass.   
 Package private members are inherited only if the subclass is present in the same package as that of superclass.   
 A superclass’s private members are not accessible outside the class itself. Rather, they’re hidden from its subclasses and can be accessed only through the non private methods inherited from the superclass.  
 Subclass can refer to members inherited from the superclass simply by using the member names. When a subclass member overrides or hides an inherited superclass member, the superclass version of the member can be accessed from the subclass by preceding the superclass member name with keyword super and a dot separator (.).  
  
**-Relationship Between Superclasses ans Subclasses** **Section 9.4.1:** class CommissionEmployee

Constructors are not inherited. However, a superclass’s constructors are still available to be called by subclasses. In fact, Java requires that the first task of any subclass constructor is to call its direct superclass’s constructor, either explicitly or implicitly (if no constructor call is specified), to ensure that the instance variables inherited from the superclass are initialized properly.

Even if a class does not have constructors, the default constructor that the compiler implicitly declares for the class will call the superclass’s default or no-argument constructor.

If you don’t explicitly specify which class a new class extends, the class extends Object implicitly.

Object’s default constructor does nothing.

-Override:A subclass can customize methods that it inherits from its superclass. To do this, the subclass overrides (redefines) the superclass method with an appropriate implementation.

@Override

public String toString()

{

return String.format("%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f",

"commission employee", firstName, lastName,

"social security number", socialSecurityNumber,

"gross sales", grossSales,

"commission rate", commissionRate);

}

To override a superclass method, a subclass must declare a method with the same signature (method name, number of parameters, parameter types and order of parameter types) as the superclass method. Also the return type must be the same as, or a subtype of the return type in the original method.

Optional(but recommended) @Override annotation indicate that the following method declaration should override an existing superclass method. This annotation helps the compiler catch a few common errors. For example signature not matching errors.

It’s a compilation error to override a method with a more restricted access modifier(making overridden method more restrictive). Doing so would break the is-a relationship, which requires that all subclass objects be able to respond to method calls made to public methods declared in the superclass.

Private methods cant be inherited. [Final methods](#final_method) [Static methods](#static_method)

**Section 9.4.2:** classBasePlusCommissionEmployee

-Why we need inheritance: We literally copied code from class CommissionEmployee and pasted it into class BasePlusCommissionEmployee, then modified class BasePlusCommissionEmployee to include a base salary and methods that manipulate the base salary. This “copy-and-paste” approach is often error prone and time consuming. Worse yet, it spreads copies of the same code throughout a system, creating code-maintenance problems—changes to the code would need to be made in multiple classes. Is there a way to “acquire” the instance variables and methods of one class in a way that makes them part of other classes without duplicating code? Next we answer this question, using a more elegant approach to building classes that emphasizes the benefits of inheritance.

**Section 9.4.3:** classBasePlusCommissionEmployee extends CommissionEmployee

This version of the program doesnt work because subclass methods cant access private instance variables of superclass.

At the design stage in an object-oriented system, you’ll often find that certain classes are closely related. You should “factor out” common instance variables and methods and place them in a superclass. Then use inheritance to develop subclasses, specializing them with capabilities beyond those inherited from the superclass.

-Explicit call to superclass constructor: We have to call one of the constructors of superclass first thing in subclass constructor. We do this by using the superclass constructor call syntax “super(argument list)”—keyword super, followed by a set of parentheses containing the superclass constructor arguments, which are used to initialize the superclass instance variables.

public BasePlusCommissionEmployee(String firstName, String lastName, String socialSecurityNumber, double grossSales, double commissionRate, double baseSalary)

{

super(firstName, lastName, socialSecurityNumber, grossSales, commissionRate);

BasePlusCommissionEmployee employee = new BasePlusCommissionEmployee(

"Bob", "Lewis", "333-33-3333", 5000, .04, 300);

System.out.printf("%n%s %s%n", "First name is", employee.getFirstName());

**Section 9.4.4:** classBasePlusCommissionEmployee extends CommissionEmployee

To reach instance methods of superclass we need to either make them protected (or package private if they are in the same package) or use public get methods. In this example we use protected.

-Subclass containts instance variables of all of its superclasses: When you create a BasePlusCommissionEmployee object, it contains all instance variables declared in the class hierarchy to that point—that is, those from classes Object (which does not have instance variables), CommissionEmployee and BasePlusCommissionEmployee. Class BasePlusCommissionEmployee does not inherit CommissionEmployee’s five-argument constructor, but explicitly invokes it (lines 14–15) to initialize the instance variables that BasePlusCommissionEmployee inherited from CommissionEmployee. Similarly, CommissionEmployee’s constructor implicitly calls class Object’s constructor. BasePlusCommissionEmployee’s constructor must explicitly call CommissionEmployee’s constructor because CommissionEmployee does not have a no-argument constructor that could be invoked implicitly.

-Problems with using protected variables instead of methods: Using protected instance variables creates several potential problems. First, the subclass object can set an inherited variable’s value directly without using a set method.

Another problem with using protected instance variables is that subclass methods are more likely to be written so that they depend on the superclass’s data implementation. In practice, subclasses should depend only on the superclass services (i.e., non-private methods) and not on the superclass data implementation. Such a class is said to be fragile or brittle, because a small change in the superclass can “break” subclass implementation. You should be able to change the superclass implementation while still providing the same services to the subclasses. [coupling\_part1](#coupling_part1)

A third problem is that a class’s protected members are visible to all classes in the same package as the class containing the protected members—this is not always desirable.

**Section 9.4.5:** class BasePlusCommissionEmployee extends CommissionEmployee  
  
Instead of using protected variables for the super class, we now use private variables and public get methods.

-Using a superclass method as part of a subclass method: If a method performs all or some of the actions needed by another method, call that method rather than duplicate its code.

@Override

public double earnings()

{

return getBaseSalary() + super.earnings();

}

@Override

public String toString()

{

return String.format("%s %s%n%s: %.2f", "base-salaried",

super.toString(), "base salary", getBaseSalary());

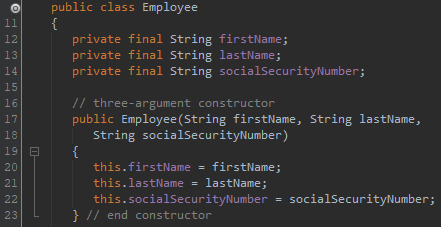
}

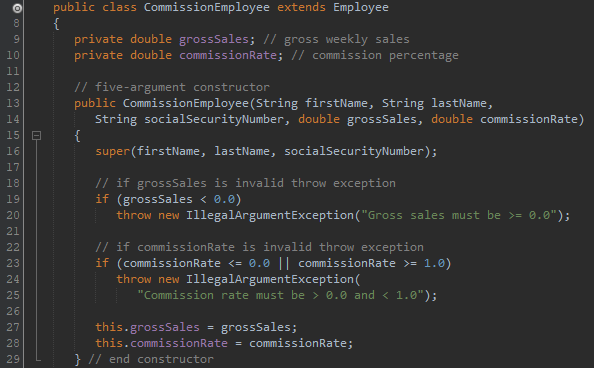
-Constructor parameter list and super parameter list:Lets think of a hierarcy of three classes. Employee, CommissionEmployee, BasePlusCommissionEmployee.

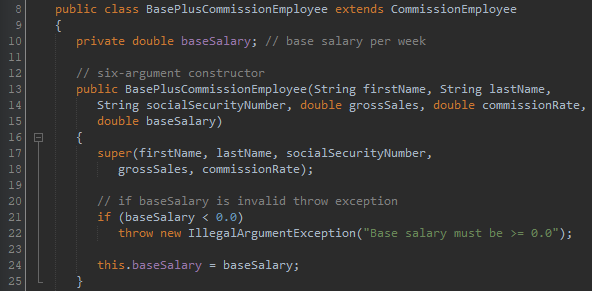
In BasePlusCommissionEmployee constructor parameter list, we list all the parameters for CommissionEmployees (firstName, lastName, socialSecurityNumber, grossSales, commissionRate) and we also list the specialized parameters of BasePlusCommissionEmployee (baseSalary). The parameters for superclass are used for the superclass constructor call “super(…)”. The parameters for subclass are used in the subclass constructor.

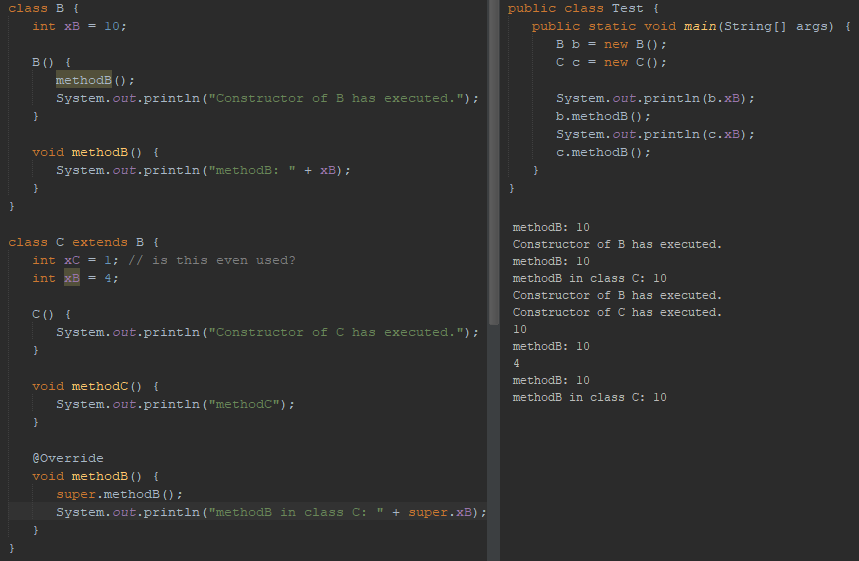
In CommissionEmployee constructor parameter list, we list all the parameters for Employees (firstName, lastName, socialSecurityNumber) and we also list the specialized parameters of CommissionEmployee (grossSales, commissionrate). The parameters for superclass are used for the superclass constructor call “super(…)”. The parameters for subclass are used in the constructor.

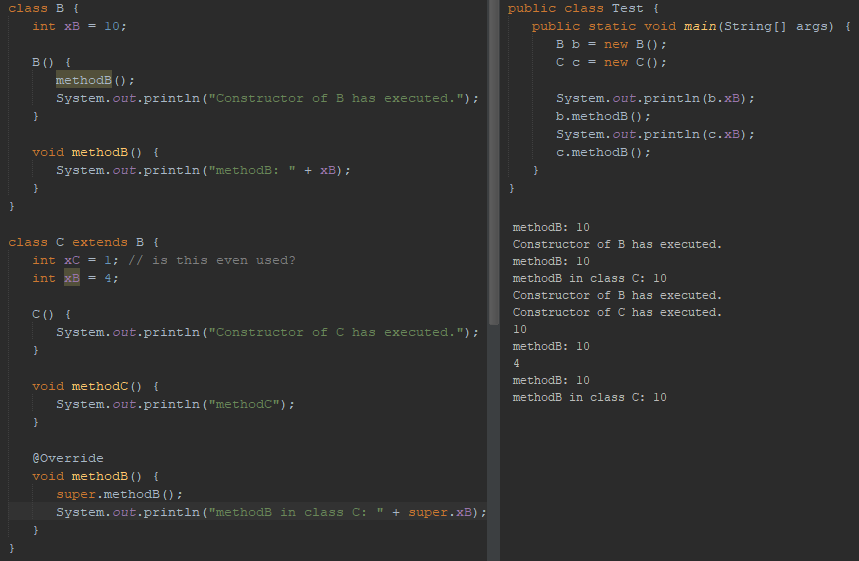
In Employee contructor parameter list, we list all the parameters of Employee.





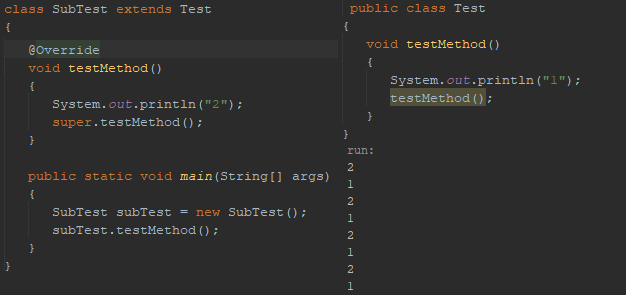


-Calling methods from superclass constructors/methods:When you create an object of a subclass, before the constructor of subclass executes, the constructor of superclass executes. And if there is a method call in superclass constructor, methods in subclass shadow methods in superclass (if subclass has overriden methods). But when fields are used in these methods, superclass fields are used, not subclass.  
  
  




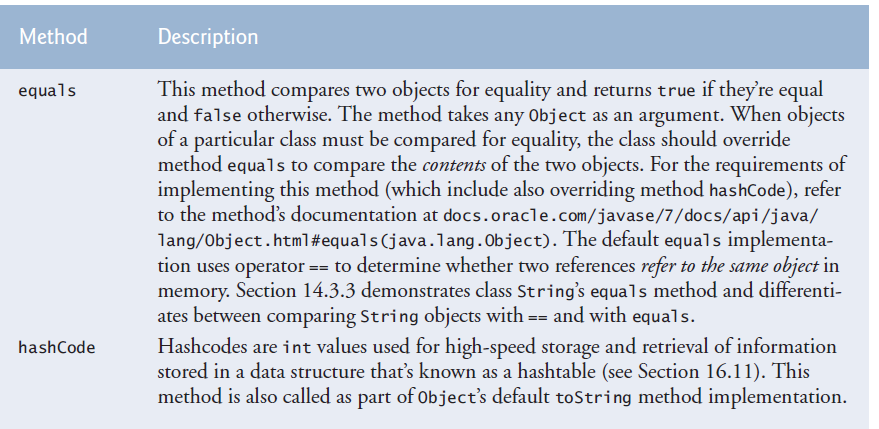
This is not just correct for constructors. If we call method of superclass from subclass and if there is another method call in superclass method, methods in subclass shadow methods in superclass (if subclass has overriden methods).

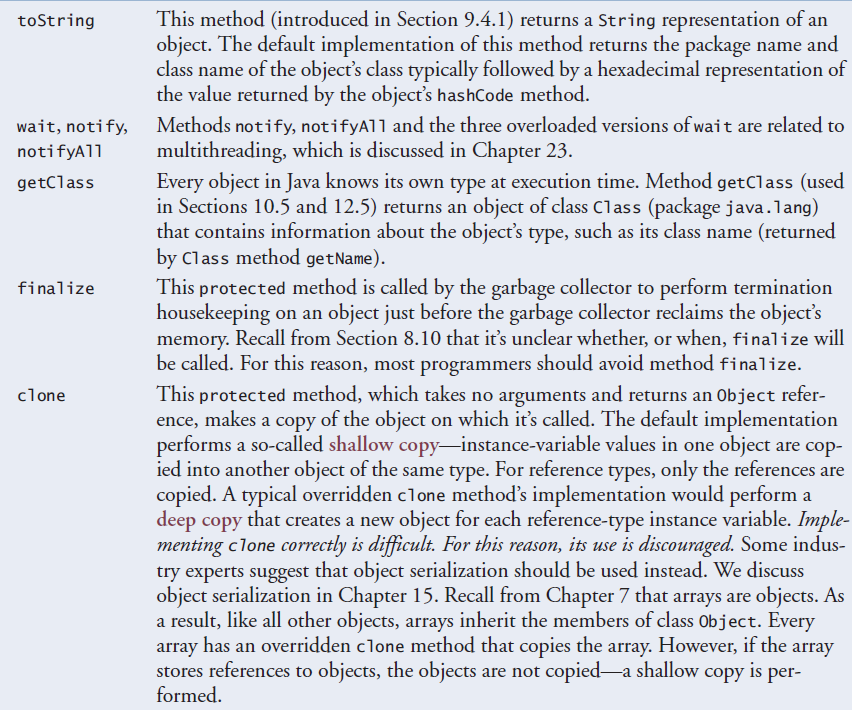
In the code below, we create an object of SubTest. We call testMethod of subTest object. “2” is printed. Then super.testMethod is called. This causes testMethod of Test class to execute. Which prints “1”. Then testMethod is called inside testMethod of Test class. But since testMethod of Test class was invoked by a subTest object, testMethod of subTest object executes.



**-Class Object:** As we discussed earlier, all classes in Java inherit directly or indirectly from class Object (java.lang), so its 11 methods (some are overloaded) are inherited by all other classes.

Class Object’s toString method is primarily a placeholder that can be overridden by a subclass to specify an appropriate String representation of the data in a subclass object.





(tag: equals, hashCode, toString, wait, notifyAll, getClass, finalize, clone, shallow copy, deep copy)

* **Object Oriented Programming: Polymorphism and Interfaces**

The same message (method) sent to a variety of objects has many forms of results—hence the term polymorphism.

The polymorphism occurs when a program invokes a method through a super class reference variable—at execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable. (dynamic method dispatch, virtual method invocation, dynamic binding, late binding) [static\_binding](#static_binding)

**-Demonstrating Polymorphic Behavior:** Invoking a method on a subclass object via a superclass reference invokes the subclass functionality. An object of a subclass can be treated as an object of its superclass.

A a = new B();

a.printIt(); // Instance method of subclass (B) executes.

A superclass reference (a) can be used to invoke only methods of the superclass (A). This superclass can inherit from its superclasses (direct and indirect) such as Object. Overridden methods in subclass (B) can be called polymorphically. Methods that are brand new in class B can’t be called. To do this, we need to downcast the reference.

Downcasting is allowed only if the object we are downcasting is an object of the class we are casting it to . Object (a) has an is-a relationship with class (B). Downcasting in the second example is allowed because “a” is-a “B”. Although it’s allowed, you should generally avoid downcasting.

A a = new B(); // Casting subclass ref to a superclass var implicit upcasting.

a.brandNewMethodInB(); // illegal

A a = new B();

B b = (B)a; // Casting superclass reference to a subclass variable. Explicit downcasting.

b.brandNewMethodInB(); // legal

If you invoke a static method, static method of the reference variable’s class will be called, not the object’s class’ because of static binding.

A a = new B();

a.printIt(); // Static method of superclass (A) executes.

**-Abstract Classes and Methods:** Sometimes it’s useful to declare classes—called abstract classes—for which you never intend to create objects. Because they’re used only as superclasses in inheritance hierarchies, we refer to them as abstract superclasses. These classes cannot be used to instantiate objects, because, as we’ll soon see, abstract classes are incomplete. Subclasses must declare the “missing pieces” to become “concrete” classes, from which you can instantiate objects. Otherwise, these subclasses, too, will be abstract.

An abstract class’s purpose is to provide an appropriate superclass from which other classes can inherit and thus share a common design.

Classes that can be used to instantiate objects are called concrete classes. Such classes provide implementations of every method they declare (some of the implementations can be inherited).

You make a class abstract by declaring it with keyword abstract. An abstract class normally contains one or more abstract methods. An abstract method is an instance method with keyword abstract in its declaration, as in

public abstract void draw(); // abstract method

Abstract methods do not provide implementations. A class that contains any abstract methods must be explicitly declared abstract even if that class contains some concrete (nonabstract) methods. Each concrete subclass of an abstract superclass also must provide concrete implementations of each of the superclass’s abstract methods.

Constructors and static methods cannot be declared abstract. Constructors are not inherited, so an abstract constructor could never be implemented. Though non-private static methods are inherited, they cannot be overridden. Since abstract methods are meant to be overridden so that they can process objects based on their types, it would not make sense to declare a static method as abstract.

**-Payroll System Using Polymorphism**

public abstract class Employee

public abstract double earnings(); // no implementation here

public class SalariedEmployee extends Employee

@Override

public double earnings() {

return getWeeklySalary();

}

public class HourlyEmployee extends Employee

@Override

public double earnings() {

if (getHours() <= 40) // no overtime

return getWage() \* getHours();

else

return 40 \* getWage() + (getHours() - 40) \* getWage() \* 1.5;

}

public class CommissionEmployee extends Employee

@Override

public double earnings() {

return getCommissionRate() \* getGrossSales();

}

public class BasePlusCommissionEmployee extends CommissionEmployee

@Override

public double earnings() {

return getBaseSalary() + super.earnings();

}

SalariedEmployee salariedEmployee =

new SalariedEmployee("John", "Smith", "111-11-1111", 800.00);

HourlyEmployee hourlyEmployee =

new HourlyEmployee("Karen", "Price", "222-22-2222", 16.75, 40);

CommissionEmployee commissionEmployee =

new CommissionEmployee("Sue", "Jones", "333-33-3333", 10000, .06);

BasePlusCommissionEmployee basePlusCommissionEmployee =

new BasePlusCommissionEmployee("Bob", "Lewis", "444-44-4444", 5000, .04, 300);

Employee[] employees = new Employee[4];

employees[0] = salariedEmployee;

employees[1] = hourlyEmployee;

employees[2] = commissionEmployee;

employees[3] = basePlusCommissionEmployee;

for (Employee currentEmployee : employees)

{

System.out.println(currentEmployee); // invokes toString

if (currentEmployee instanceof BasePlusCommissionEmployee)

{

BasePlusCommissionEmployee employee = (BasePlusCommissionEmployee) currentEmployee;

employee.setBaseSalary(1.10 \* employee.getBaseSalary());

System.out.printf("new base salary with 10%% increase is: $%,.2f%n",

employee.getBaseSalary());

}

System.out.printf("earned $%,.2f%n%n", currentEmployee.earnings());

}

for (int j = 0; j < employees.length; j++)

System.out.printf("Employee %d is a %s%n", j,

employees[j].getClass().getName());

**instanceof:** The condition is true if object referenced by the lvalue is an rvalue. This would also be true for any object of a rvalue subclass because of the is-a relationship a subclass has with its superclass. (tag: instance of)

**-A Deeper Explanation of Issues with Calling Methods from Constructors:** Recall that when you construct a subclass object, its constructor first calls one of the direct superclass’s constructors. If the superclass constructor calls an overridable method(instance method), the subclass’s version of that method will be called by the superclass constructor—before the subclass constructor’s body has a chance to execute. This could lead to subtle, difficult-to detect errors if the subclass method that was called depends on initialization that has not yet been performed in the subclass constructor’s body.

It’s acceptable to call a static method from a constructor. For example, a constructor and a set method often perform the same validation for a particular instance variable. If the validation code is brief, it’s acceptible to duplicate it in the constructor and the set method. If lengthier validation is required, define a static validation method (typically a private helper method) then call it from the constructor and the set method. It’s also acceptable for a constructor to call a final instance method, provided that the method does not directly or indirectly call an overridable instance method.

**-Interfaces:** An interface declaration begins with the keyword interface and contains only constants and abstract methods. All methods declared in an interface are implicitly public abstract methods, and all fields are implicitly public, static, and final.

Interfaces can’t be instantiated. They dont have constructors.

An interface has to be declared in a .java file with the same name as the interface. Interfaces are compiled into .class files.

To use an interface, a concrete class must specify that it implements the interface and must declare each method in the interface with the signature specified in the interface declaration. To specify that a class implements an interface, add the implements keyword and the name of the interface to the end of your class declaration’s first line. A class that does not implement all the methods of the interface is an abstract class and must be declared abstract.

An interface is often used when disparate classes—i.e., classes that are not related by a class hierarchy—need to share common methods and constants. This allows objects of unrelated classes to be processed polymorphically.

public interface Payable

{

double getPaymentAmount(); // calculate payment; no implementation

}

public class Invoice implements Payable

@Override

public double getPaymentAmount()

{

return getQuantity() \* getPricePerItem(); // calculate total cost

}

Classes can inherit only one class but classes and interfaces can implement multiple interfaces.

All objects of a class that implements interfaces have the is-a relationship with each implemented interface type.  
  
 public class ClassName extends SuperClassName implements FirstInterface,

SecondInterface, …

Objects of any subclasses of the class that implements the interface can also be thought of as objects of the interface type.

public abstract class Employee implements Payable

public class SalariedEmployee extends Employee

// implement interface Payable method that was abstract in superclass Employee

@Override

public double getPaymentAmount()

{

return getWeeklySalary();

}

Payable[] payableObjects = new Payable[4];

payableObjects[0] = new Invoice("01234", "seat", 2, 375.00);

payableObjects[1] = new Invoice("56789", "tire", 4, 79.95);

payableObjects[2] = new SalariedEmployee("John", "Smith", "111-11-1111", 800.00);

payableObjects[3] = new SalariedEmployee("Lisa", "Barnes", "888-88-8888", 1200.00);

System.out.println("Invoices and Employees processed polymorphically:");

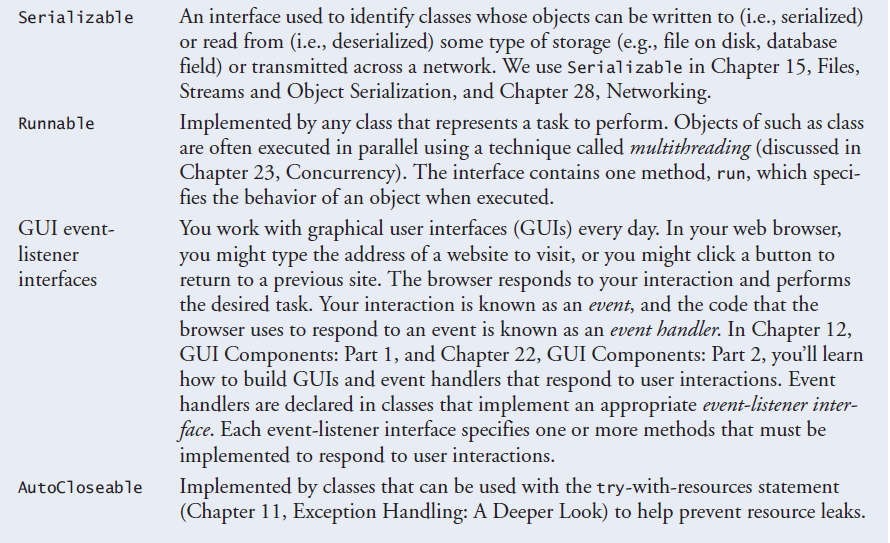
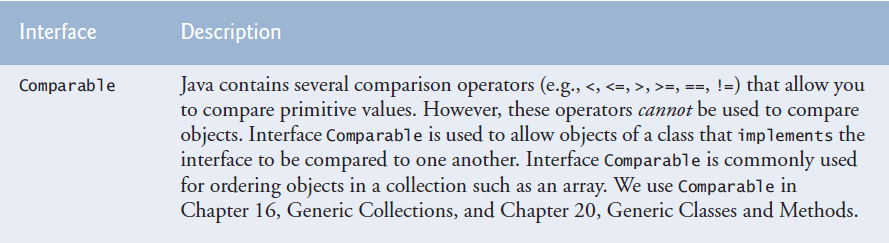
for (Payable currentPayable : payableObjects)

{

System.out.printf("%n%s %n%s: $%,.2f%n", currentPayable.toString(), "payment

due", currentPayable.getPaymentAmount());

}

**-Some Common Interfaces of the Java API:** Many of the Java API methods take interface arguments and return interface values.  
 

(tag: Comparable, Serializable, Runnable, GUI event-listener interfaces, GUI event listener interfaces, AutoCloseable)

**-Java SE 8 Interface Enhancements**

**-default Interface Methods:** In Java SE 8, interfaces also may contain public default methods with concrete default implementations that specify how operations are performed when an implementing class does not override the methods. If a class implements such an interface, the class also receives the interface’s default implementations (if any). To declare a default method, place the keyword default before the method’s return type and provide a concrete method implementation.

Any class that implements the original interface will not break when a default method is added—the class simply receives the new default method.

Prior to Java SE 8, adding methods to an interface would break any implementing classes that did not implement the new methods. If you didn’t want to change the old code, you could have created another interface and made it extend the previous interface. Then your users had the option to either use the old interface or upgrade to the new interface.

**-static Interface Methods:** Prior to Java SE 8, it was common to associate with an interface a class containing static helper methods for working with objects that implemented the interface. With static interface methods, such static helper methods can now be declared directly in interfaces rather than in separate classes.

**-Functional Interfaces:** As of Java SE 8, any interface containing only one abstract method is known as a functional interface. Some examples of functional interfaces are ActionListener, Comparator, and Runnable.

**-Four main principles of object oriented programming**

**-Encapsulation (Information hiding):** Packaging related data and methods, and not sharing hidden data with outside directly. Methods are used to access hidden data.

**-Abstraction:** Hiding implementation details, and only providing functionality to the user. In Java, abstraction is achieved using abstract classes and interfaces.

**-Inheritance:** Reuse the common logic and extract the unique logic into a separate class.

**-Polymorphism:** Gives a way to use a class exactly like its parent. This allows us to use subclasses in a list/array of superclass. Objects of superclass and all its subclasses can respond to common superclass methods. Can also be done with an interface and all the classes that implement it.

**-SOLID Principles**

**-Single-responsiblity principle:** A class should have only one job.  
  
 **-Open-closed principle:** Entities should be open for extension, but closed for modification. We do this by moving the logic to subclasses, and we call the logic of subclasses from this method through a superclass array. Which means we can extend this method to use new subclasses without modifying it.

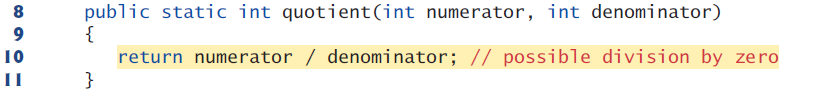
**-Liskov substitution principle:** Every subclass should besubstitutable for its superclass.

**-Interface segregation principle:** A client should never be forced to implement an interface that it doesn’t use or clients shouldn’t be forced to depend on methods they do not use.

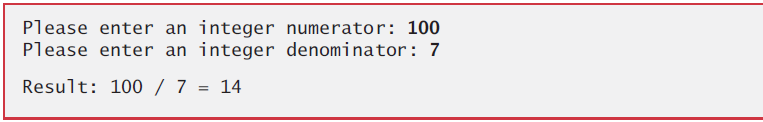
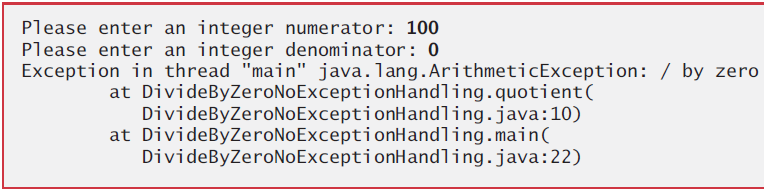
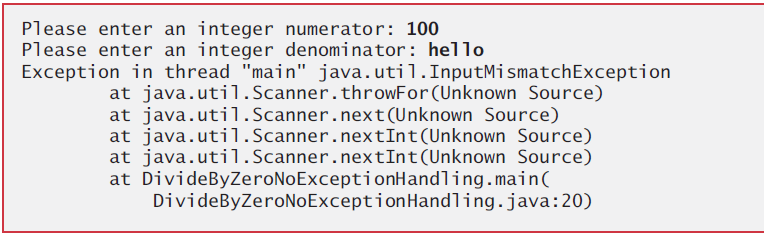
**-Dependency Inversion Principle:** Program to an interface/superclass, not to an implementation. Just like Open-Closed principle.

* **Exception Handling: A Deeper Look** [exception handling part2](#exception_handling_part2)

**-Without exception handling:** Exceptions are thrown (i.e., the exception occurs) by a method when it detects a problem and is unable to handle it.





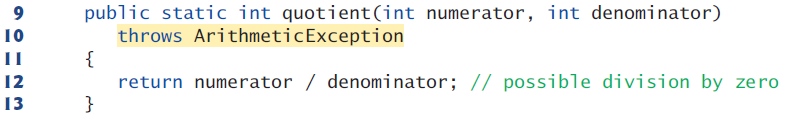
  
  
  
  


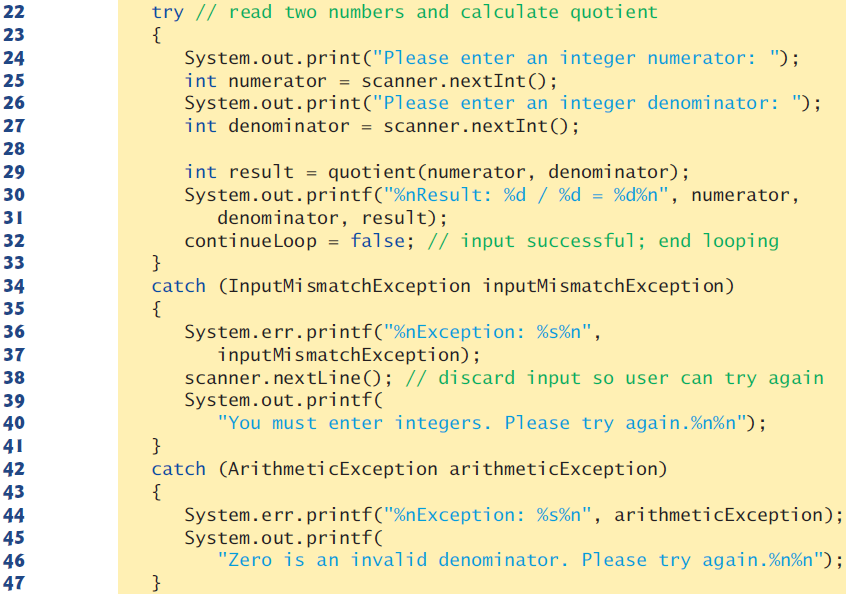
Several lines of information are displayed in response to invalid input (divide by zero, enter string instead of int). This information is known as a stack trace, which includes the name of the exception (java.lang.ArithmeticException) in a descriptive message that indicates the problem that occurred and the method-call stack (i.e., the call chain) at the time it occurred. The stack trace includes the path of execution that led to the exception method by method. This helps you debug the program.

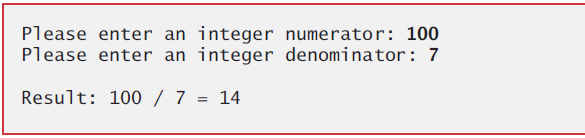
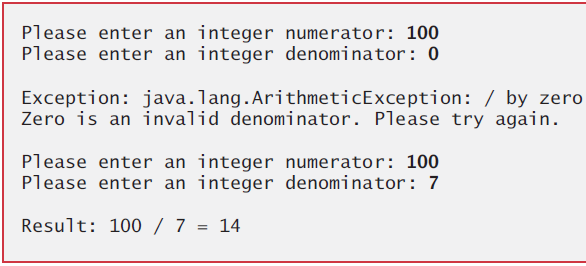
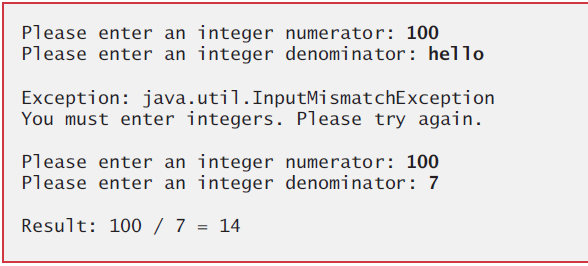
Each line of the stack trace contains the class name and method (e.g., DivideByZeroNoExceptionHandling.main) followed by the filename and line number (e.g., DivideByZeroNoExceptionHadling.java:22).

The top row of the call chain indicates the throw point—the initial point at which the exception occured.

Sometimes a program may continue even though an exception has occurred and a stack trace has been printed. In such cases, the application may produce unexpected results. For example, a graphical user interface (GUI) application will often continue executing.

**-With exception handling**  
  




  
  
  
  
  
  
Try block encloses the code that might throw an exception and the code that should not execute if an exception occurs (i.e., if an exception occurs, the remaining code in the try block will be skipped).

Catch block (also called catch clause or exception handler) catches (i.e., receives) and handles an exception.

At least one catch block or a finally block must immediately follow the try block. Each catch block specifies in parentheses an exception parameter that identifies the exception type the handler can process. When an exception occurs in a try block, the catch block that executes is the first one whose type matches the type of the exception that occurred (i.e., the type in the catch block matches the thrown exception type exactly or is a direct or indirect superclass of it).

We use the System.err (standart error stream) object to output error messages. By default, System.err’s print methods, like those of System.out, display data to the command prompt.

System.out and System.err are streams—sequences of bytes. While System.out (known as the standard output stream) displays a program’s output, System.err (known as the standard error stream) displays a program’s errors. Output from these streams can be redirected (i.e., sent to somewhere other than the command prompt, such as to a file). Using two different streams enables you to easily separate error messages from other output. [What's wrong with using System.err](https://stackoverflow.com/questions/1049795/whats-wrong-with-using-system-err-in-java) [streams part2](#streams_part2) [streams\_part4](#streams_part4)

It’s relatively common for a try block to be followed by several catch blocks to handle various types of exceptions. If the bodies of several catch blocks are identical, you can use the multi-catch feature. (tag: multi catch)

catch (*Type1* | *Type2* | *Type3* e)

Java uses a “multithreaded” model of program execution—each thread is a concurrent activity. One program can have many threads. If a program has only one thread, an uncaught exception will cause the program to terminate. If a program has multiple threads, an uncaught exception will terminate only the thread in which the exception occurred. In such programs, however, certain threads may rely on others, and if one thread terminates due to an uncaught exception, there may be adverse effects on the rest of the program.

If an exception occurs in a try block the try block terminates immediately and program control transfers to the first of the following catch blocks in which the exception parameter’s type matches the thrown exception’s type.

After the exception is handled, program control does not return to the throw point, because the try block has expired (and its local variables have been lost). Rather, control resumes after the last catch block. This is known as the termination model of exception handling. Some languages use the resumption model of exception handling, in which, after an exception is handled, control resumes just after the throw point.

If no exceptions are thrown in the try block, the catch blocks are skipped and control continues with the first statement after the catch blocks.

The try block and its corresponding catch and/or finally blocks form a try statement.

Throws clause specifies the exceptions the method might throw if problems occur. This clause, which must appear after the method’s parameter list and before the body, contains a comma-separated list of the exception types. Such exceptions may be thrown by statements in the method’s body or by methods called from there.

Read the online API documentation for a method before using it in a program. The documentation specifies the exceptions thrown by the method (if any) and indicates reasons why such exceptions may occur. Next, read the online API documentation for the specified exception classes. The documentation for an exception class typically contains potential reasons that such exceptions occur. Finally, provide for handling those exceptions in your program.

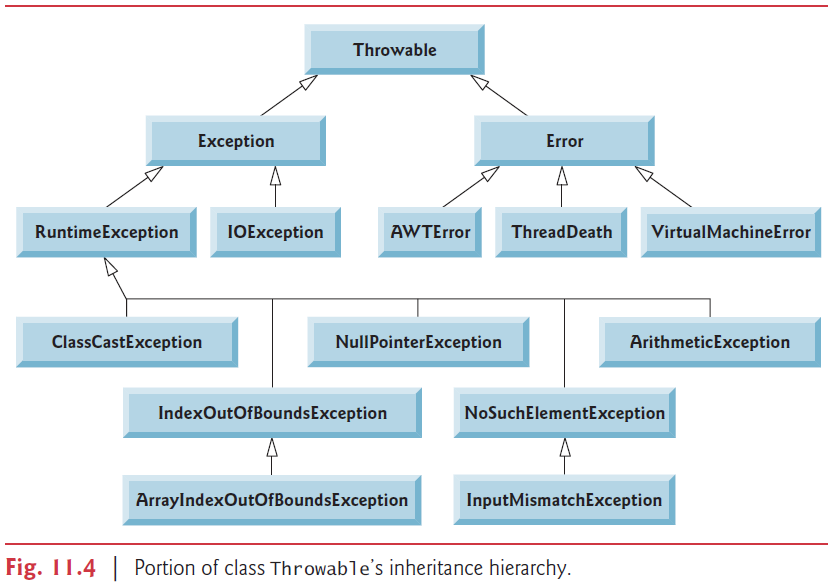
When a method throws an exception, it terminates and does not return a value, and method’s local variables go out of scope (and are destroyed). If method contained local variables that were references to objects and there were no other references to those objects, the objects would be marked for garbage collection.

**-When to use exception handling:** Exception handling is designed to process synchronous errors, which occur when a statement executes. Exception handling is not designed to process problems associated with asynchronous events (e.g., disk I/O completions, network message arrivals mouse clicks and keystrokes), which occur in parallel with, and independent of, the program’s flow of control.

Exception handling provides a single, uniform technique for documenting, detecting and recovering from errors. This helps programmers working on large projects understand each other’s error-processing code.

**-Java exception hierarchy:** Only Throwable objects can be used with the exception-handling mechanism.

Class Exception and its subclasses represent exceptional situations that can occur in a Java program and that can be caught by the application. Class Error and its subclasses represent abnormal situations that happen in the JVM. Most Errors happen infrequently and should not be caught by applications—it’s usually not possible for applications to recover from Errors.



[Class Throwable](https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html) [Java built-in exceptions](https://www.tutorialspoint.com/java/java_builtin_exceptions.htm)

All exception types that are direct or indirect subclasses of RuntimeException (java.lang) are unchecked exceptions. These are typically caused by defects in your program’s code.

Classes that inherit directly or indirectly from class Error are unchecked, because Errors are such serious problems that your program should not even attempt to deal with them.  
  
Unlike checked exceptions, the Java compiler does not examine the code to determine whether an unchecked exception is caught or declared. Unchecked exceptions typically can be prevented by proper coding.

It is recommended that you provide exception-handling when proper coding can’t prevent unchecked exceptions such as NumberFormatException from Integer method parseInt.

All classes that inherit from class Exception but not directly or indirectly from class RuntimeException are considered to be checked exceptions. Such exceptions are typically caused by conditions that are not under the control of the program—for example, in file processing, the program can’t open a file if it does not exist.

The compiler checks each method call and method declaration to determine whether the method throws a checked exception. If so, the compiler verifies that the checked exception is caught or is declared in a throws clause—this is known as the catch-or-declare requirement.

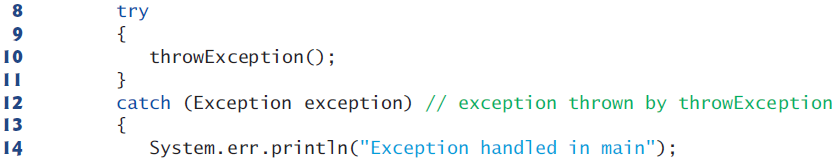
If a subclass method overrides a superclass method, it’s an error for the subclass method to list more exceptions in its throws clause than the superclass method does. However, a subclass’s throws clause can contain a subset of a superclass’s throws clause.

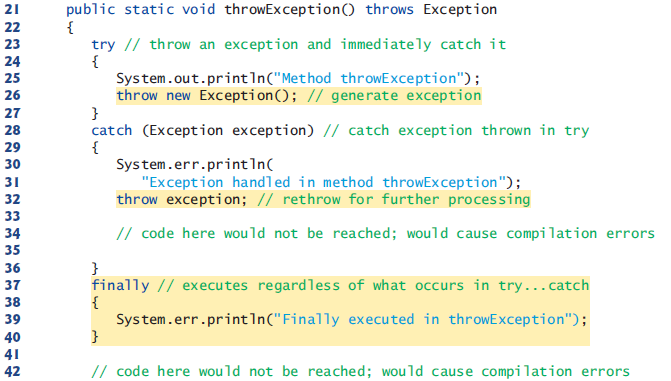
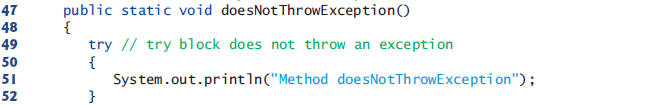
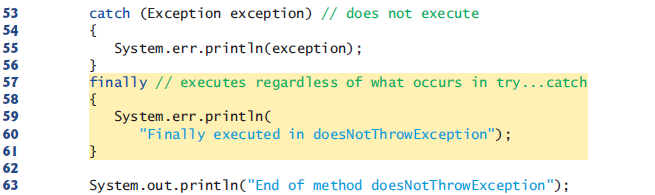
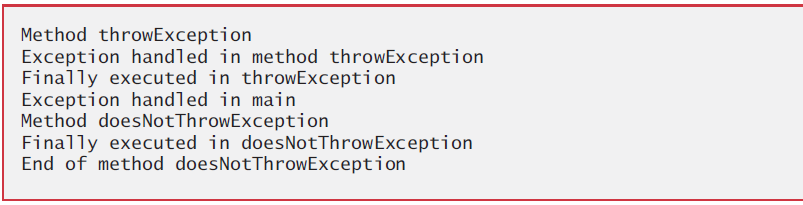
If your method calls other methods that throw checked exceptions, those exceptions must be caught or declared. If an exception can be handled meaningfully in a method, the method should catch the exception rather than declare it.  
  
**-Finally block (finally clause):** The finally block will execute whether or not an exception is thrown in the corresponding try block. The finally block also will execute if a try block exits by using a return, break or continue statement or simply by reaching its closing right brace. The one case in which the finally block will not execute is if the application exits early from a try block by calling method System.exit.

If an exception that occurs in a try block cannot be caught by one of that try block’s catch handlers, the program skips the rest of the try block and control proceeds to the finally block. Then the program passes the exception to the next outer try block—normally in the calling method—where an associated catch block might catch it. This process can occur through many levels of try blocks. Also, the exception could go uncaught.

If a catch block throws an exception, the finally block still executes.

When an exception is thrown from a catch or finally block and not caught in that block, the previous exception is lost and the lastest exception will be returned.  
  
Because a finally block always executes, it typically contains resource-release code. Resources such as objects, files, database connections and network connections.



You can throw exceptions yourself by using the throw statement.

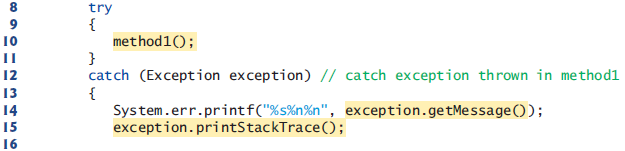
Throw exceptions from constructors to indicate that the constructor parameters are not valid—this prevents an object from being created in an invalid state.

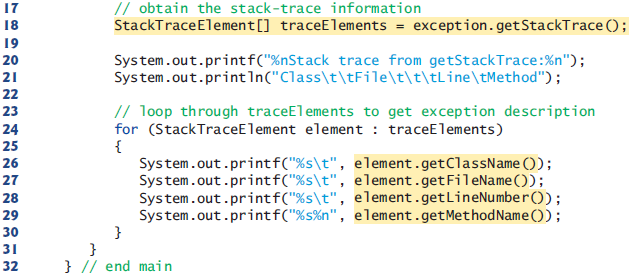
Rethrowing the exception is when a catch block, upon receiving an exception, decides either that it cannot process that exception or that it can only partially process it.

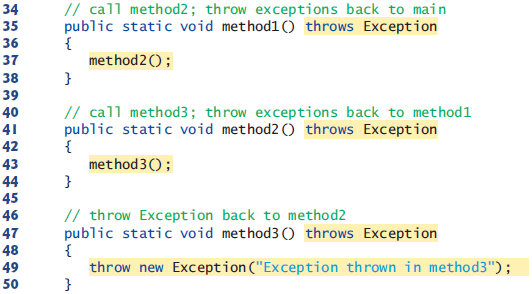
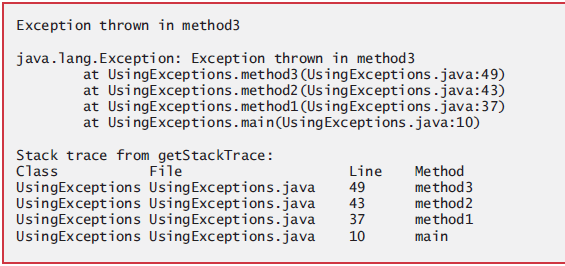
Avoid placing in a finally block code that can throw an exception. If such code is required,

enclose the code in a try…catch within the finally block.

**-Stack unwinding and ontaining information from an exception object:** When an exception is thrown but not caught in a particular scope, the method-call stack is “unwound,” and an attempt is made to catch the exception in the next outer try block. This process is called stack unwinding. Unwinding the method-call stack means that the method in which the exception was not caught terminates, all local variables in that method go out of scope and control returns to the statement that originally invoked that method. If a try block encloses that statement, an attempt is made to catch the exception. If a try block does not enclose that statement or if the exception is not caught, stack unwinding occurs again.



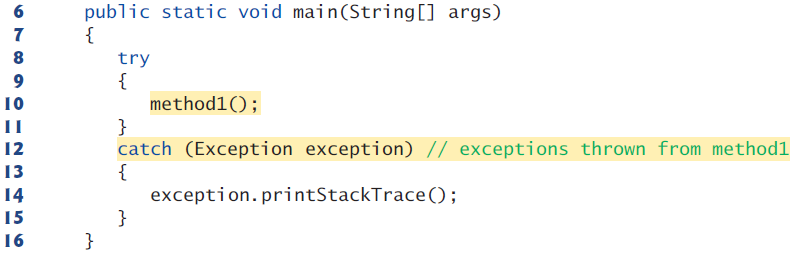
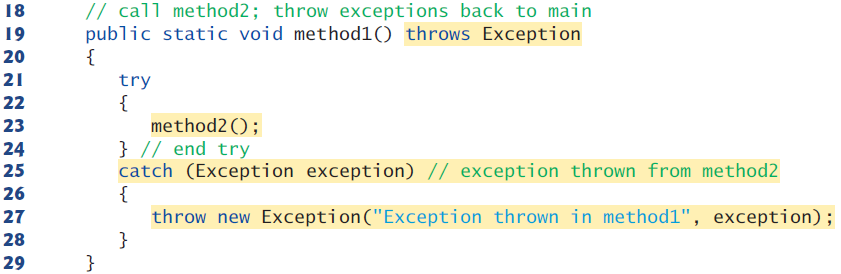


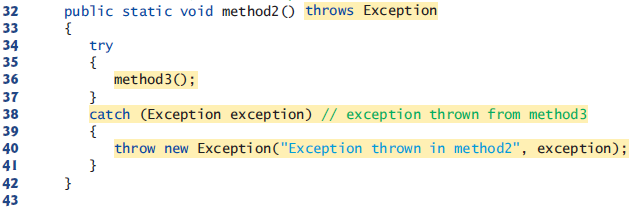
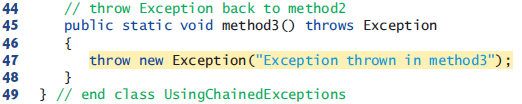
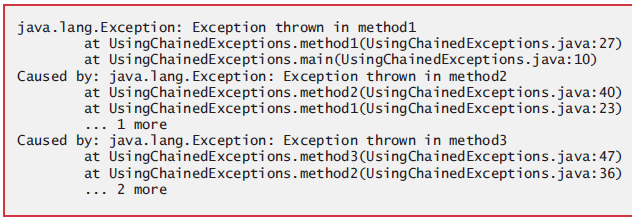
  
  


Throwable method toString returns a String containing the name of the exception’s class and a descriptive message.

Class Throwable’s getMessage method returns the descriptive string stored in an exception.

Class Throwable has a printStackTrace method that outputs to the standard error stream the stack trace. Often this is helpful in testing and debugging. Class Throwable also provides a getStackTrace method that retrieves the stack-trace information that might be printed by printStackTrace.  
 An exception that’s not caught in an application causes Java’s default exception handler to run. This displays the name of the exception, a descriptive message that indicates the problem that occurred and a complete execution stack trace.

**-Chained exceptions:** Sometimes a method responds to an exception by throwing a different exception type that’s specific to the current application. If a catch block throws a new exception, the original exception’s information and stack trace are lost. Earlier Java versions provided no mechanism to wrap the original exception information with the new exception’s information to provide a complete stack trace showing where the original problem occurred. This made debugging such problems particularly difficult. Chained exceptions enable an exception object to maintain the complete stack-trace information from the original exception.  
  
  
  


**-Declaring new exception types:** A typical new exception class contains only four constructors:

1) one that takes no arguments and passes a default error message String to the superclass

constructor

2) one that receives a customized error message as a String and passes it to the superclass

constructor

3) one that receives a customized error message as a String and a Throwable (for

chaining exceptions) and passes both to the superclass constructor

4) one that receives a Throwable (for chaining exceptions) and passes it to the superclass

constructor.

When defining your own exception type, study the existing exception classes in the Java API and try to extend a related exception class.   
 If the existing classes are not appropriate superclasses for your new exception class, decide whether your new class should be a checked or an unchecked exception class.

**-Preconditions and postconditions:** Programmers spend significant amounts of time maintaining and debugging code. To facilitate these tasks and to improve the overall design, you can specify the expected states before and after a method’s execution. These states are called preconditions and postconditions, respectively.

A precondition must be true when a method is invoked. Preconditions describe constraints on method parameters and any other expectations the method has about the current state of a program just before it begins executing. If the preconditions are not met, then the method’s behavior is undefined.

A postcondition is true after the method successfully returns. Postconditions describe constraints on the return value and any other side effects the method may have.

Typically, a method’s preconditions and postconditions are described as part of its specification. When designing your own methods, you should state the preconditions and postconditions in a comment before the method declaration.

**-Assertions:** When implementing and debugging a class, it’s sometimes useful to state conditions that should be true at a particular point in a method. These conditions, called assertions, help ensure a program’s validity by catching potential bugs and identifying possible logic errors during development. Preconditions and postconditions are two types of assertions.

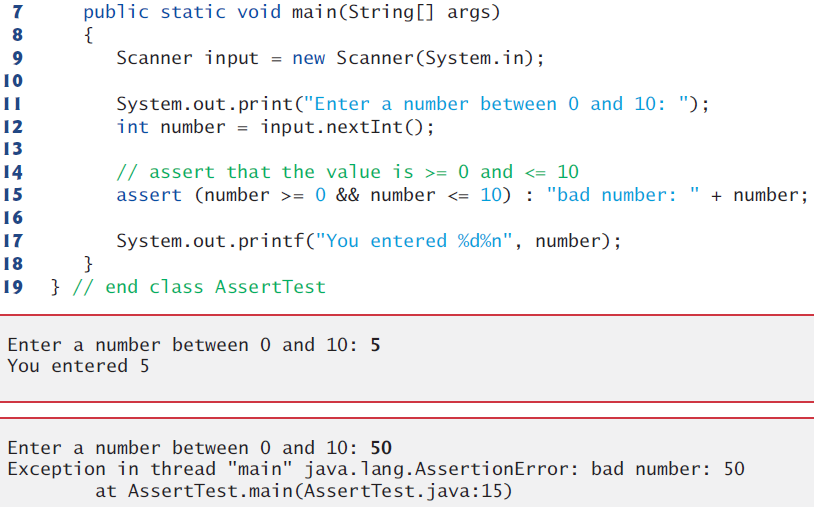
Java includes two versions of the assert statement for validating assertions programatically. The assert statement evaluates a boolean expression and, if false, throws an AssertionError (a subclass of Error). The first form of the assert statement is

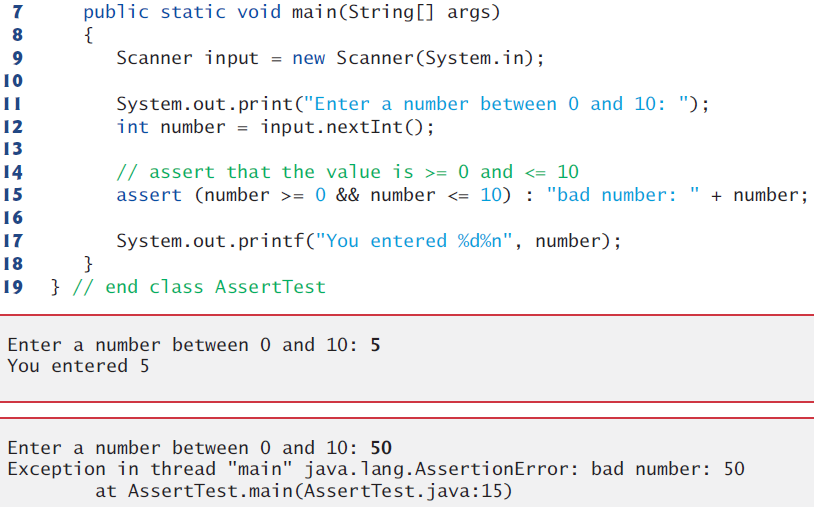
assert *expression*;

which throws an AssertionError if expression is false. The second form is

assert *expression1* : *expression2*;

which evaluates expression1 and throws an AssertionError with expression2 as the error message if expression1 is false.



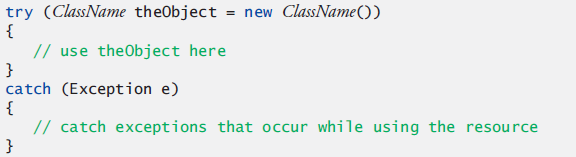


You must explicitly enable assertions when executing a program, because they reduce performance and are unnecessary for the program’s user. To do so, use the java command’s -ea command-line option, as in

Java -ea AssertTest

Use assertions only during development, not in production code.

**-Try-with-resources: Automatic Resource Deallocation:** An alternative notation to releasing resources in finally block is the try-with-resources statement.  
 Each resource must be an object of a class that implements the Auto-Closeable interface, and thus provides a close method. (tag: try with resources)



The try-with-resources statement implicitly calls the theObject’s close method at the end of the try block. You can allocate multiple resources in the parentheses following try by separating them with a semicolon (;).

* **GUI Components**

-IDE:You can select multiple elements and select options such as align, anchor, etc.  
 Right-click a component, events, select an event to code logic for.

Swing has a cross-platform look-and-feel known as Nimbus.  
  
**-Simple GUI-Based Input/Output with JOptionPane** (javax.swing.JoptionPane):

JOptionPane.showMessageDialog(rootPane, "Hello World");

GUI applications are made of windows and dialog boxes (also called dialogs). Message dialog will show a message. The first argument refers to the parent window and causes the dialog to appear centered over the app’s window. If the first argument is null, the dialog box is displayed at the center of your screen. The second argument is the String to display in the dialog box.  
  
 String name = JOptionPane.showInputDialog(rootPane, "What is your name");

String message = String.format("Welcome, %s, to Java Programming!", name);

JOptionPane.showMessageDialog(rootPane, message);

Input dialog has a prompt and a field (known as text field)where user can enter text. Clicking the OK or pressing Enter returns the string to program and saves it in the string “name”. Clicking the Cancel or pressing Esc returns null and the program displays the word “null” as the name.

String firstNumber = JOptionPane.showInputDialog("Enter integer");

int number1 = Integer.parseInt(firstNumber);

JOptionPane.showMessageDialog(null, "Number is " + number1);

Third argument for dialogs is title. Fourth argument is messageType. Here are the message types: ERROR\_MESSAGE, INFORMATION\_MESSAGE, WARNING\_MESSAGE, QUESTION\_MESSAGE, PLAIN\_MESSAGE.

A JoptionPane dialog is a modal dialog. While the dialog is on the screen, the user cannot interact with the rest of the application.  
 Do not overuse modal dialogs, as they can reduce the usability of your applications. Use a modal dialog only when it’s necessary to prevent users from interacting with the rest of an application until they dismiss the dialog.  
­

**-Overview of Swing Components:** AWT components look like the native GUI components of the platform on which a Java program executes.  
 Swing GUI components allow you to specify a uniform look-and-feel for your application across all platforms or to use each platform’s custom look-and-feel.

Most Swing components are lightweight components—they’re written, manipulated and displayed completely in Java. AWT components are heavyweight components, because they rely on the local platform’s windowing system to determine their functionality and their look-and-feel.  
  
Class Component (java.awt) is a superclass that declares the common features of GUI components in packages java.awt and javax.swing. Any object that is a Container (subclass of Component) (java.awt) can be used to organize Components by attaching the Components to the Container. Containers can be placed in other Containers to organize a GUI.   
 Class JComponent (javax.swing) is a subclass of Container. JComponent is the superclass of all lightweight Swing components and declares their common attributes and behaviors.  
  
**-Displaying Text and Images in a Window:** A JLabel (label) displays text, image or both.  
  
Since we specify only a filename, Java assumes that the image is in the same directory as the class. ImageIcon supports formats such as GIF, JPEG and PNG.

statusBar.setText(String.format("Clicked at [%d, %d]", evt.getX(), evt.getY()));  
 **-Text Fields and an Introduction to Event Handling with Nested Classes:** GUIs are event driven. When the user interacts with a GUI component, the interaction— known as an event—drives the program to perform a task.  
 The code that performs a task in response to an event is called an event handler, and the process of responding to events is known as event handling.  
  
-textFieldActionPerformed:  
  
 // Pressing enter in studentNumber textfield puts focus on password textfield  
 password.requestFocus();  
  
 // Pressing enter in password passwordfield invokes manualSubmit button.   
 manualSubmitActionPerformed(evt);   
  
 // We can use jTextField1.getText instead of evt.getActionCommand.

String string = String.format("textField1: %s", evt.getActionCommand());

JOptionPane.showMessageDialog(rootPane, string);  
  
Top-level classes are not declared inside another class. Classes inside other classes are called nested classes. Nested classes can be static or non-static. Non-static nested classes are called inner classes.  
 An inner-class object must be created by an object of its top-level class. Each inner-class object implicitly has a reference to an object of its top-level class. The inner-class object is allowed to use this implicit reference to directly access all the variables and methods of the top-level class. A nested class that’s static does not require an object of its top-level class and does not implicitly have a reference to an object of the top-level class. [nested\_class\_part1](#nested_class_part1)  
 When used in an inner class, keyword this refers to the current inner-class object being manipulated. An inner-class method can use its outer-class object’s this by preceding this with the outer-class name and a dot (.) separator, as in ButtonFrame.this.

An anonymous inner class is a class that’s declared without a name and typically appears inside a method declaration. An anonymous inner class only has access to the final local variables of the method in which it’s declared. Since an anonymous inner class has no name, one object of the class must be created at the point where the class is declared.

imagesJComboBox.addItemListener(new ItemListener() {

...

});

**-Adapter Classes:** For many event-listeners that contain multiple methods, you dont have to declare every method in your anonymous inner class. An adapter class provides a default implementation (with an empty method body) of each method that we can override.

**-Common GUI Event Types and Listener Interfaces:** The event source is the GUI component with which the user interacts. The event object (of a class that extends java.awt.AWTEvent) encapsulates information about the event that occurred, such as a reference to the event source and any event-specific information that may be required by the event listener for it to handle the event. The event listener is an object that’s notified by the event source when an event occurs; in effect, it “listens” for an event, and one of its methods executes in response to the event. A method of the event listener receives an event object when the event listener is notified of the event. The event listener then uses the event object to respond to the event. This event-handling model is known as the delegation event model—an event’s processing is delegated to an object (the event listener) in the application.   
 For each event-object type, there’s typically a corresponding event-listener interface. An event listener for a GUI event is an object of a class that implements one or more of the event-listener interfaces from packages java.awt.event and javax.swing.event. When an event occurs, the GUI component with which the user interacted notifies its registered listeners by calling each listener’s appropriate event-handling method.  
  
**-Buttons That Maintain State:** JToggleButton, JCheckBox, and JRadioButton have the isSelected method and can be added to a buttonGroup.  
  
 Font font = null; // stores the new Font

if (boldJCheckBox.isSelected() && italicJCheckBox.isSelected())

font = new Font("Serif", Font.BOLD + Font.ITALIC, 14);  
 jTextField1.setFont(font);

**-JComboBox:** JComboBoxis a generic class. When you create a JComboBox, you specify the type of the objects that it manages—the JComboBox then displays a String representation of each object.

Some useful methods for combo boxes are getItemCount, addItem, getSelectedItem.

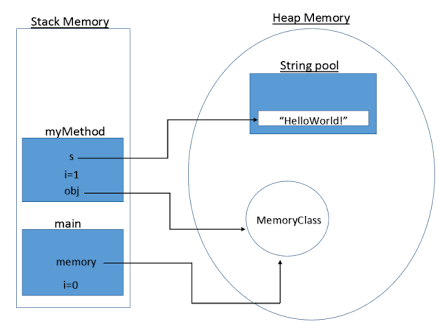
Using the model property, you can either hard code values or you can use the custom code to get values from another source. Both combo box and list use the code below.  
  
 new javax.swing.DefaultComboBoxModel(names)We can use getClass().getResource() to get the URL of a resource.  
  
**-JList:** JListis a generic classand it supports both single and multiple selection lists.  
 When we are using a single selection list, we code the logic into the list. But when we use a multiple selection list, we code the logic into another component such as a button.

copyJList.setListData(colorJList.getSelectedValuesList().toArray(new String[0]));  
  
Each JFrame actually consists of three layers—the background, the content pane and the glass pane. The content pane appears in front of the background and is where the GUI components in the JFrame are displayed. The glass pane is used to display tool tips and other items that should appear in front of the GUI components on the screen.   
 **-Mouse Event Handling:** evt.isMetaDown for right click. evt.isAltDown for middle mouse button.

**-Key Event Handling:** KeyEvent.getKeyText(evt.getKeyCode()) // used for pressed and released  
  
 evt.getKeyChar() // used for typed  
  
**-JtextArea:** Like multiple-selection Jlists, we code the logic into another component such as a button.  
 Using the text property, you can either hard code values or you can use the custom code (string reference for both text field and text area) to get values from another source.

* **Strings, Characters and Regular Expressions**

String (character string, string literal) objects in Java are immutable—they cannot be modified after that are created.  
  
Strings are reference types but act like primitive types in that they can be initialized like primitive type variables.  
  
-Stack and heap of Strings:If we initialize strings without using new, string is kept in string pool. In string pool you can have only one copy of one string such as “test1” so you dont waste memory. All strings that are equal to this, point to it. If we use new to create strings, a string is created in heap, outside of string pool. Every new string created with new has its own memory even if they are the same string. String pool is in heap in its own memory. [String intern()](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#intern())



Ways to assign a string

1) String string1 = "Hello World!";

2) String string1 = new String(“java”);

3) String string1 = new String(array);

String string1 = new String(array1, 6, 3); // use 3 characters starting with 6th

Ways to assign char arrays

1) char[] array1 = new char[10];  
  
2) char[] array1 = {'b', 'i', 'r', 't', 'h', ' ', 'd', 'a', 'y'};

**-String.format():** Returns a formatted string. [Class String](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html)

String string1 = String.format("No: %s String: \"%s\"", 3 \* x, string2);

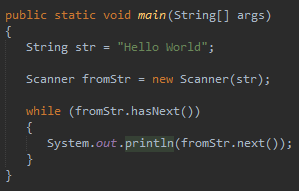
We can do the same thing using string concatenation.

String string1 = "No: " + 3 \* x + " String: \"" + string2 + "\"";

**-Notes on String object methods**

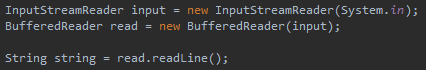
**-Equals:** Always put the value/variable you know for sure is a string to the left so the program doesn’t break.   
 == operator checks whether two objects are same (same reference). It doesnt check whether two objects are equal (contents).

-**split:** hasNext or StringTokenizer can be used for the same effect. [string\_tokenizer](#string_tokenizer)

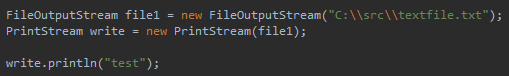
String[] words = string1.split(" ")  
  
 

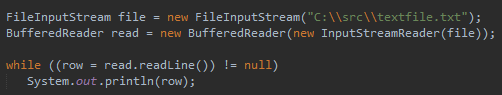
* **Advanced I/O and Serialization** (java.io.\*) [streams\_part3](#streams_part3)

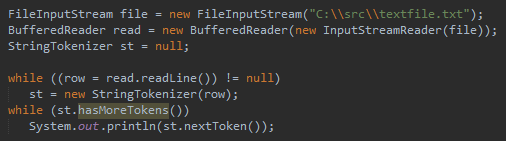
**-Read from user**

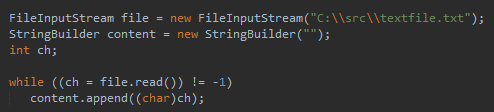


**-Write to file**

  
  
**-Read rows from file**

  
  
**-Read words from file**

  
  
**-Read characters from file**



* **Accessing Databases With JDBC(Java Database Connectivity):**

Table name words should be seperated by underscores.

Column name words should be seperated by underscores and be all uppercase.