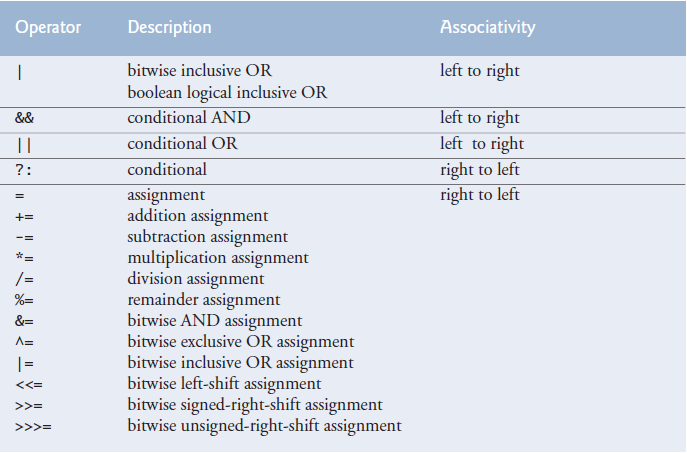
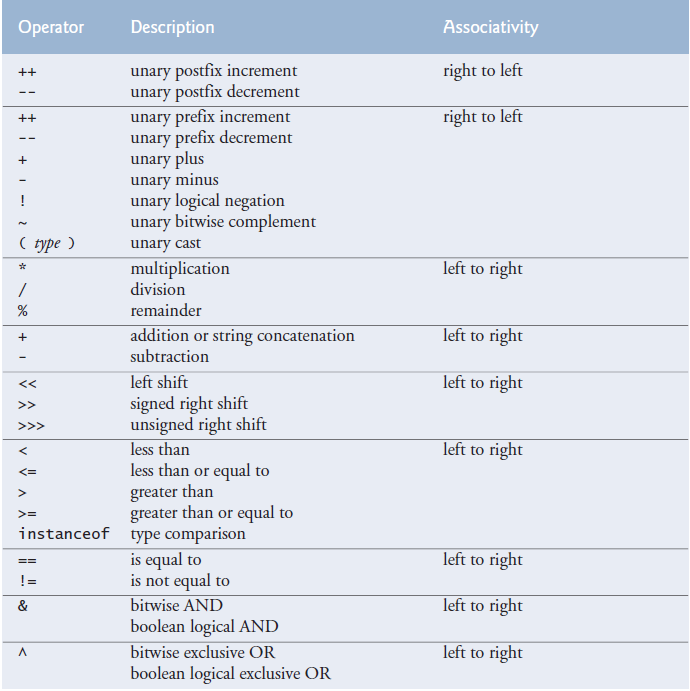
**Part 1**

* **Operators**

  
Operator precedence goes from high to low in the table. Associativity tells us which operator to execute first when there are multiple operators of the same precedence.  
 You can use parentheses to force the order of evaluation.  
 **-Operators based on operand number:** A unary operator contains one operand “++a”. A binary operatorcontains two operands: “a = b”. The ternary operator contains three operands [conditional\_operator](#conditional_operator)  
  
-Unary minus: x = 16 - - 5 -> 16 - (-5) -> x = 16 + 5 // x = 16 --5; (Error)  
  
We can use “x % y” and “x / y” to get rid of/gather certain digits. [Programming Tricks & Algorithms.docx](../../y-%20Computer%20Documents/Programming%20Tricks%20&%20Algorithms.docx#getting_rid_of_gathering)  
  
The 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.  
  
**-Bitwise operators** You can add more notes from the web appendice

int a = 60; // 60 = 0011 1100

int b = 13; // 13 = 0000 1101

int c = 0; // 0 = 0000 0000

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

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

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

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

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

c = a >> 2; // 15 = 0000 1111. Uses 1 or 0 based on left most bit.

c = a >>> 2; // 15 = 0000 1111. Uses 0.

**-Shorthand operators** are a shorter way to express a statement already available in a language.

**-Compound/Composite assignment operators:** += -= \*= /= %= operators are compound operators. (tag: compound assignment operators)

x = x + 2;

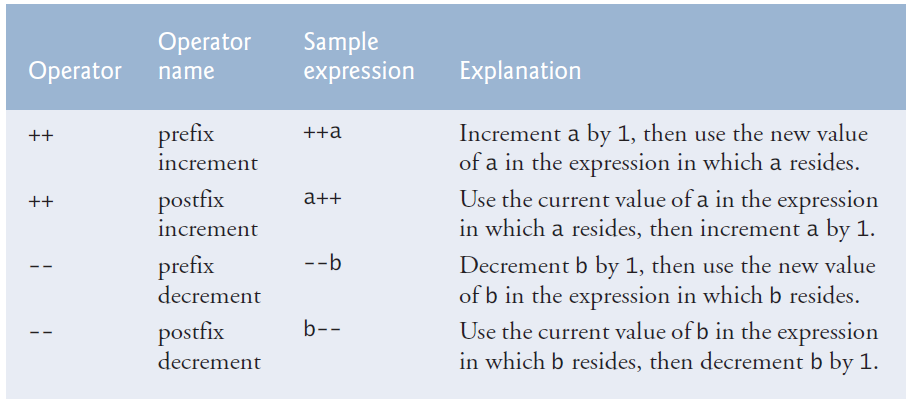
x += 2;

-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** add or subtract 1 from the value of a numeric variable.



(tag: preincrementing**,** predecrementing, postincrementing,postdecrementing)

When incrementing or decrementing a variable in a statement by itself, the prefix and postfix forms have the *same* effect. It’s only when a variable appears in the context of a larger expression that prefix and postfix forms have different effects.

-Three ways of incrementing/decrementing

passes = passes + 1;

passes += 1;

++passes; // passes++;

* **Data Types**

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 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. When you are calculating floating-point values, the number can get so small that it overflows and becomes negative.

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

double a = 1e-3;

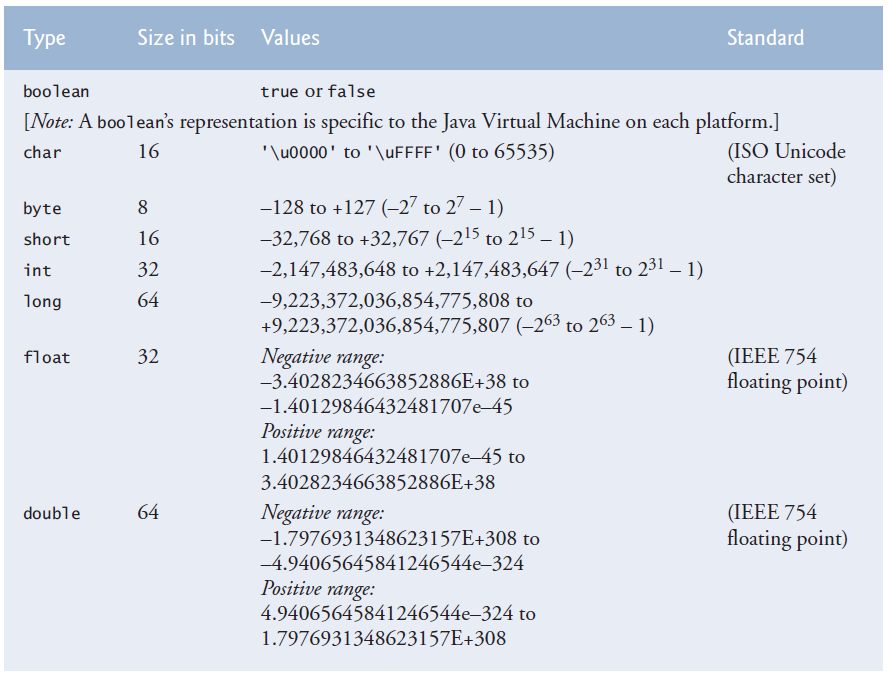
float b = 1e+3f; // + is optional  
  
Float variables areSingle-precision (32 bits) floating-point numbers. Can hold up to 7 significant digits. Double variables are double-precision (64 bits) floating-point numbers. Double requires twice the memory but can hold 15 significant digits. Float and double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

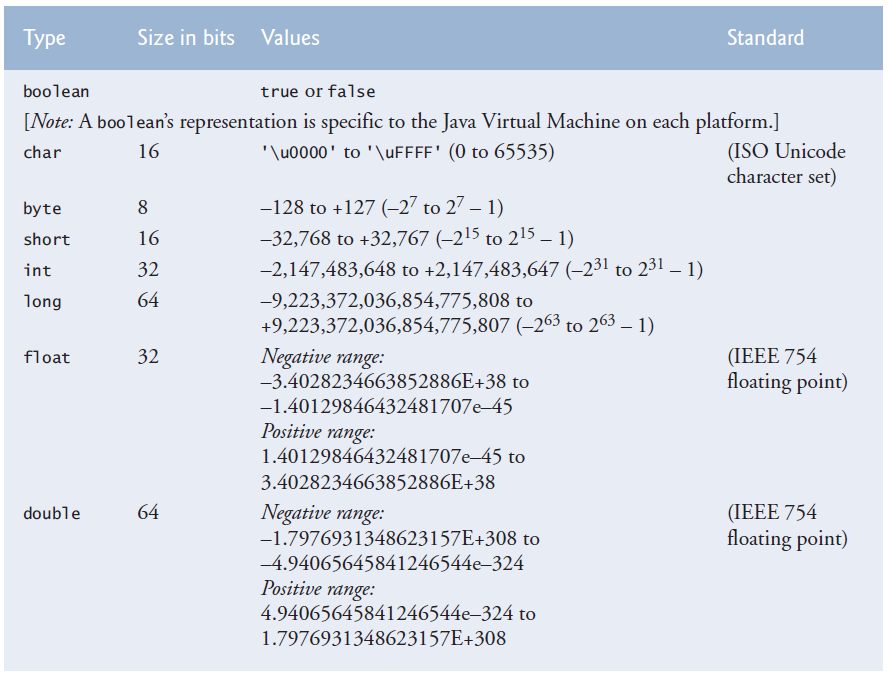
Floating-point numbers are not %100 precise. We have float, double and BigDecimal depending on our needs.

Maximum and minimum values that can be stored in primitive variables are represented by constants MIN\_VALUE and MAX\_VALUE, which are defined in corresponding class types of each primitive type in package java.lang.

char can represent letters, digits, special characters (\*, %), escape sequences (\n). chars are held as integers in the computer.  
 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. ASCII character set is a subset of the Unicode character set used by Java. Java uses UTF-16 character encoding.

char unicode1 = 'a', unicode2 = 'π', unicode3 = '\u0041';







**-Default values**

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

Integer literals are considered int, and floating-point literals are considered double.

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

float myVariable = 9;  
  
Trying to assign a double literal to a float variable, which requires an explicit type casting.

float myVariable = 9.5; // Won’t compile  
  
Explicitly casting double literal to a float literal and assigning it to float variable.

float myVariable = (float) 9.5;  
  
Assigning 9.5 float literal to a float variable.

float myVariable = 9.5F;

Suffixes are only needed for literal values. Not needed for reading a value from a user since you use methods like nextFloat, which doesn’t reject the 9.5 double value because it has implicit cast.

**-Data Type Conversions** [argument\_promotion\_and\_casting](#argument_promotion_and_casting)

**-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 are not allowed to make an assignment that would cause data loss or 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 6/4 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**

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.

Value of 21 + 1 is known in compile time. But value of b1 + 1 is not known in compile time.  
  
 byte b1 = 21; // Making this final would fix the error  
byte b3 = b1 + 1; // Compiler error. Possible lossy conversion from int to byte.  
byte b4 = 21 + 1; // Compiles, implicit cast. "-129 +1" also compiles.

**­-String <=> Primitive:** You can also use %o and %x [conversion\_characters](#conversion_characters)

String string = Integer.toString(number); // toString(number, radix)

String string = String.valueOf(number); // Returns null instead of an exception  
  
int number = Integer.parseInt(string); // parseInt(string, radix);

int number = Integer.valueOf(string); // valueOf(string, radix);

**-Literals in different bases**

-int literals in different bases: Precede them with a minus to make them negative.

// Decimal declaration. Possible chars are [0-9]

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

// Octal declaration starts with 0. Possible chars are [0-7]

// Binary representation starts with 0B or 0b. Possible chars are [0-1]

int decimal = 495, hexa = 0x1EF, octal = 0757, binary = 0b111101111;

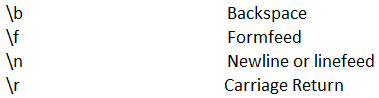
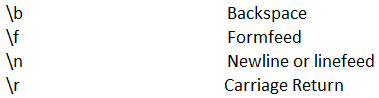
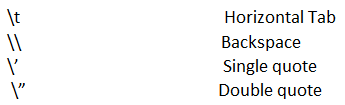
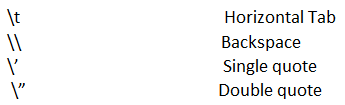
-char literals in different bases: char octal = '\ooo', hexadecimal = '\uhhhh';

**-Unicode to char and String**

short code = 65;

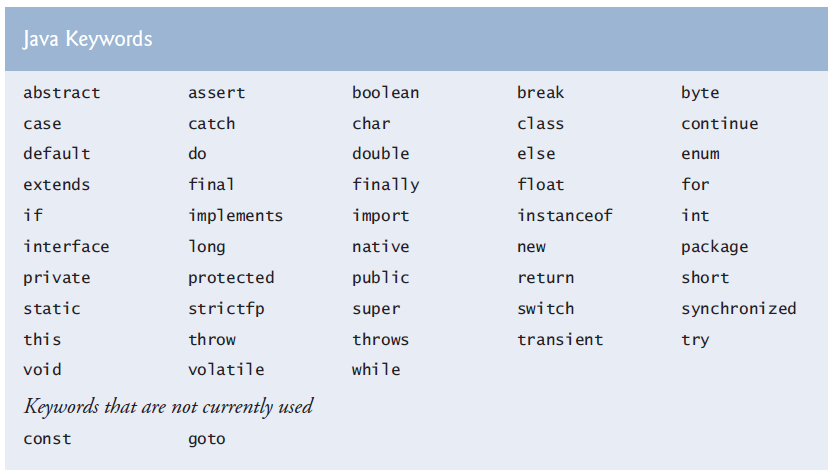
char c1 = (char) code;

String unicodeString = Character.toString(c1);

**-Control characters, escape characters, escape sequences:** “\” is the escape character. \n is an escape sequence. When a backslash appears in a string (variable, literal (including) 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. Cannot be used as identifiers



Java also contains the reserved words true and false (boolean literals), null (no reference literal).

Fixed values in source code are literals (3.75, 3, ‘z’, “z\t").

* **Programming Style / Conventions**

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

**-Identifier:** The names we give to packages, classes, enums, interfaces, methods, variables and 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: [Conventions](https://www.oracle.com/java/technologies/javase/codeconventions-contents.html)

Starting with a letter. The dollar sign is not used at all. Capitalizing the first letter of the words after the first one is camelCase, capitalizing the first letter too is PascalCase.

Package identifiers should be all lowercase.

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 lowercase 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.

List static variables first, then instance variables. Declare one variable per line. Use good identifiers and comments to make the program self-documented, readable and understandable.

Setter and getter methods start with words set and get. Predicate method names begin with “is”.  
  
The main method argument is named "args" or "argv".   
  
When using if else structures don’t indent “else if” or “else”.  
  
Put a space between control statements and their conditions “if (x > 10)” but don’t put a space between method names and their argument list “factorial(x);”

The unary operators should be placed next to their operands, with no intervening spaces.  
  
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.

Put one empty line after a block if there is more code under it.

* **Terms**

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

Expression:Combination of variables, constants, method calls, and operators which computes to a value.

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

if (number1 == number2); //if (number1 == number2) { };  
  
**-Declaration, Assignment, Initialization** int x; // declaration  
 x = 10; // assignment  
int x = 10; // initializtion  
  
**-White space characters:** Newline (\n), Space, Tab (\t). Ignored by the compiler.

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

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

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

-**Javadoc comments** enable you to embed program documentation directly in your programs. The Javadoc utility program generates HTML documentation using 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. The tokens are: Keywords, Identifiers, Literals, Operators, Separators. The separators are:() {} [] ; , . :

***Part 3***

* **Package**

Predefined 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 a class to a package:  
  
package packageName;

**-Import:** If you look at the 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, go to properties of your project, go to “Libraries”, 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 are added to your project, they become reachable with their fully qualified names. We can import their content 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 don’t need to import classes if you 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. Including the predecessor of all classes (Object class). Classes in the same package are also implicitly imported.   
  
Importing classes from the default package is a compile-time error. Because it has no name.

APIs of old Java versions are called deprecated and might be removed from newer 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)

* **Console Input/Output**

**-System.out** object is known as the standard output object/stream. “out” is a static reference variable of type PrintStream in System class. [streams\_part2](#streams_part2) [Class System](https://docs.oracle.com/javase/7/docs/api/java/lang/System.html)

**-print, println, printf:** One string can’t span more than one line of code but you can divide a string into multiple lines and concatenate those strings.

Print displays a line of text. Println puts a newline after printing. Printf prints formatted strings. We can print formatted without printf, using concatenation.

Printf’s first argument is a format string that may consist of fixed text, format specifiers,andcontrol 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.

An output that leads the user to take an action is called a prompt.

Newline Character and the line separator:%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)

-Formatting big numbers to make them easier to read: You can make big numbers more readable by using underscores. This works 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)); // .replace(",", " "));  
  
**-String concatenation**You can to assemble String objects into larger strings by using operators +, +=, or concat(String).

+ takes the string or non-string literal/variable on one side and merges it with the string literal/variable on the other side, in a new temporary string [implicit\_toString](#implicit_toString).

+= takes the string variable on the left side and the string or non-string literal/variable on the right side and merges them in a new temporary string and assigns it to the variable on the left.

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.

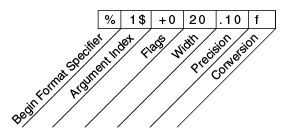
A boolean concatenated with a String is converted to the String "true" or "false".

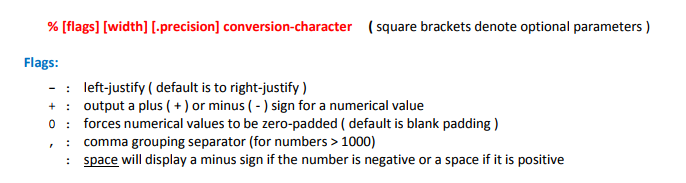
Confusing the + operator used for string concatenation with the + operator used for addition can lead to strange results. Java evaluates the operands of “addition or string concatenation” operator, from left to right. For example, if integer variable y has the value 5, the expression

"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".

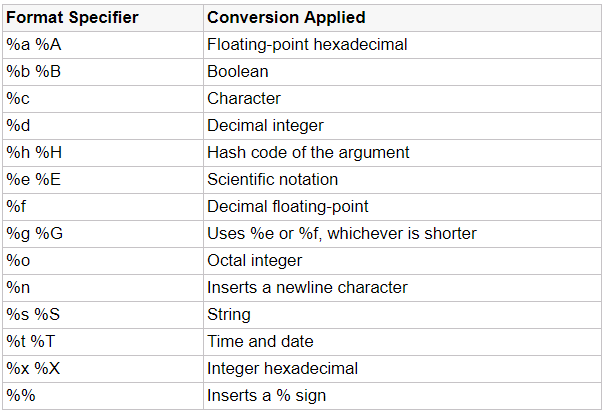
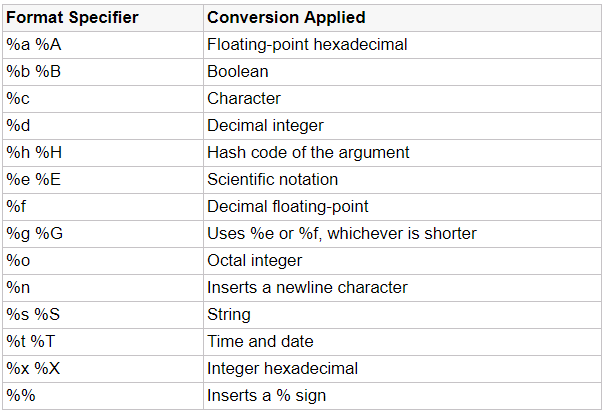
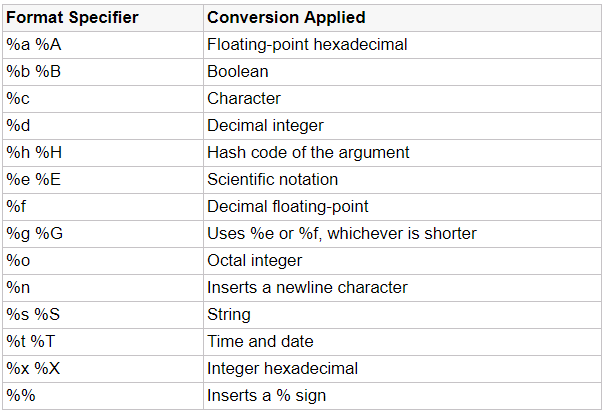
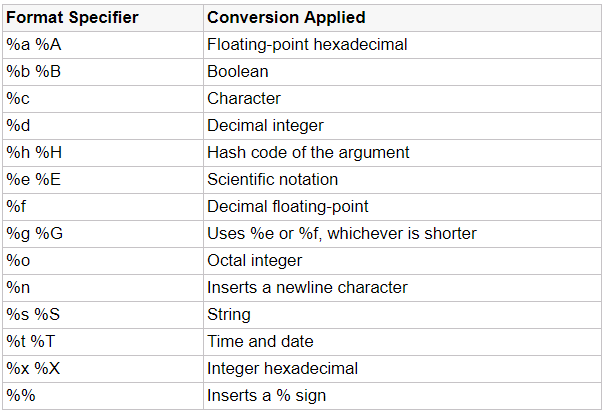
**-Formatting Output**-Format specifier





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

-Conversion characters



-Printing tables: [Programming Tricks & Algorithms.docx](../../y-%20Computer%20Documents/Programming%20Tricks%20&%20Algorithms.docx#printing_tables)

If the value to be output were more than field width character positions wide, the field width would be extended to the right to accommodate the entire value.

The actual separator used is specific to the user’s locale.

When using precision, the number is *rounded* to the nearest number precision allows. 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](-%20Other/Java%20API%20Notes.docx#dynamic_precision_part2) We don’t have the %\*.\*f from C.  
  
 String.format(String.format("%%.%ds", precision), "foo");

-A Warning about Displaying Rounded Values: 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 of 32.907, which would normally be rounded to 32.91 for display purposes. Thus, your output could appear as 14.23 + 18.67 = 32.91. But a person adding the individual numbers as displayed would expect the sum to be 32.90.

Do not use double (or float) variables to perform precise monetary calculations. The imprecision of floating-point numbers can lead to errors. Use java.math.BigDecimal instead.  
  
**-System.in** object is known as the standard input object/stream. “in” is a static reference variable of type InputStream in System class. [streams\_part1](#streams_part1) [streams\_part3](#streams_part3)

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

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

-**nextLine:** Characters are read by the method until it encounters a newline and returns a string containing characters up to but not including newline, which is discarded.

-Extra newline character: Pressing enter puts a newline character after the characters you typed. Non-nextLine readers don’t absorb the newline character. If you use a non-nextLine, you need to absorb its newline character with a nextLine method. Then you can use next or nextLine.

You don’t 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.

int x = input.nextInt();

input.nextLine(); // Can use "input.hasNextLine()" after to see if there is more.

String s = input.nextLine(); // Can use "s.isEmpty()" after to see if you read text

-**next** starts reading with the first non white space character and reads 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.

You don’t have to assign an input to a variable to use that value. You can use an input directly.

if (input.nextInt() == 0)

**-EOF (End Of File):** C works with EOF, newline character, and white space character much more since C works with characters. Java works with lines and words so you don’t “keep reading characters until EOF (or null), newline character, or whitespace character is found”. Instead you “as long as there are more lines or words, keep reading them”.  
  
The end-of-file indicator (EOF)is a system-dependent keystroke combination that 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** determines whether there is more input data. As long as the end-of-file indicator has not been reached, hasNext will return true.   
 Has methods do not consume the value. There are other other has methods such as nextInt.

while (input.hasNext()) {

int grade = input.nextInt();

total += grade;

}

* **Introduction to Object Oriented Programming**

If we have a public class in a .java file, that .java file must be named after that public class “ClassName.java”. There can only be one public class in a .java file.

[driver\_classes](#driver_classes)

**-Creating Classes and objects:** You can declare new classes to fit your needs. That is why Java is an extensible language. Each new class you create becomes a new type that can be used to declare variables and create objects. Creating an object of a class is called instantiation. Objects are instances of classes.  
  
 **-New:** The keyword new allocates memory for the object (requests 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 a 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 value. A variable’s name enables the program to access the value of the variable in memory. A Destructive process is a process that causes some data to be lost (writing). A Non-destructive process is a process that doesn’t cause data to be lost (reading).

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

**-Reference type** variables (called references) store addresses of objects in the 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.

The size of reference variables are generally 32 bit in 32 bit systems and 64 bit in 64 bit systems.

referenceVariable = null; // Doesn’t 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.

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

-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.

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

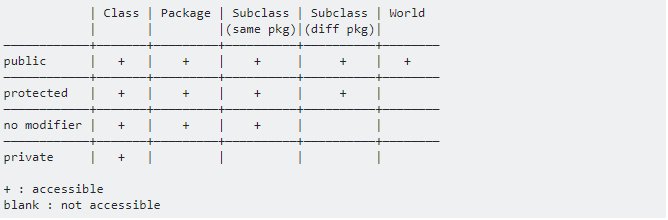
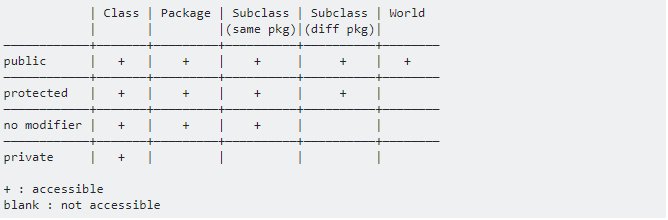
Declared inside a class but outside methods.

Their scope is the entire body of the class. Their access however depends on the access modifier. Most instance variables are private so the variable can only be used by methods of its own class.

They are used to represent attributes of a class.

You don’t have to initialize them. They have default values.

Fields exist as long as the object exists.

**-Local Variable**  
  
 Declared directly inside a method or inside a block in a method.  
 [scope\_of\_a\_declaration](#scope_of_a_declaration). Each block can access only its local variables. In the case of nested blocks, inner blocks can use local variables of outer blocks.  
 They are used to store temporary information.   
 You have to initialize them. They don’t have default values.  
 Local variables exist until their scope ends.  
  
 **-Arguments** are the values we use while invoking a method and are passed to parameters.  
  
 **-Parameters** are declared in the parameter list. They are initialized with arguments. Argument number has to be equal to parameter number. Types don’t have to be the same 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 the access to classes, methods, and fields.  
  
World: Other packages in the project.  
  
No modifier is also called package private or package access.  
  
Public methods are also called public services or public interfaces. 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.  
  
-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 and access 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 non-private access to instance fields. You might think set and get methods don’t 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. Instance variables are encapsulated (hidden, and related data and methods are bundled)in the object and can be accessed only by methods of the object.  
  
-Setter:Commonly called mutator methods. Usually setter methods have void return type.  
  
-Getter:Commonly called accessor methods or query methods. A get method that returns a boolean value is commonly called a predicate method.  
  
**-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 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.  
 Non-static methods are typically called instance methods. [single method in memory](#single_method_in_memory)

Static methods are implicitly final [final\_methods](#final_methods). You can hide the superclass static method by just making a brand new static method with the same signature. To use the static method of the superclass, use SuperclassName.methodName. (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.

-Static Class: To have a static class, make the class final, make the constructor private, make all members static. [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/)

Nested classes can be actually static. Instantiating static nested class from static context,   
  
  OuterClass.NestedClass nestedObject = new OuterClass.NestedClass();

Instantiating inner class from static context, [nested\_class\_part2](#nested_class_part2)  
  
 OuterClass outerObject = new OuterClass();

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

-Static initializer: 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 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)

public class Test {

// static variables

static { ... }

public static void main(String[] args) { ... }

}

**-this, variable shadowing:** It is sometimes called this reference.You canuse the field 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 or Classname.fieldName to access the field.

The second use of this reference is accessing constructors using this().  
  
Every object can access a reference to itself with keyword this [single\_method\_in\_memory](#single_method_in_memory).

We can’t use this reference from a static context.

**-Constructor:** Java requires a constructor call for every object that is created. If you don’t create a constructor, the compiler provides an implicit constructor. If you declare a constructor then there is no default constructor, which means you can’t create an object with the default constructor “new Account()” unless you create one yourself.

Constructors can’t have return types, not even void. Normally, constructors are declared public.

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

[calling\_methods\_from\_constructors](#calling_methods_from_constructors)

* **Control Statements and Structured Programming**

**-Algorithm:** [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)

**-Condition** is an expression that can be true or false. Expressions you can use as a condition are true, false, a boolean variable, a simple condition, or a complex condition.

Control 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 using the logical operators &&, ||, &, |, ^, and !.

**-Logical Operators**

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

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

-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

if ((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).

In expressions using operator &&, a 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—modification of a variable’s value in a condition.

if ((birthday == true) | (++age >= 65))

For clarity, avoid expressions with side effects (such as assignments) in conditions.

**-Boolean Logical Exclusive OR (^):** A 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.

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

with an appropriate relational or equality operator. “if (!(grade == sentinelValue))” can be written as “if (grade != sentinelValue)”.

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

**-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 come 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.

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.

**-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 the controlling expression of type char, String, byte, short, int, or enum.

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 *groups of actions*). The switch statement is called a multiple-selection statement because it selects among many different actions (or groups of actions).  
  
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.

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).  
  
-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.

**-Counter-controlled repetition** uses 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 repetitions 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 shouldn’t use them as counters. Use integers.

**-Sentinel-controlled 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.” Sentinel-controlled repetition is often called indefinite repetition because 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 a sentinel controlled loop, prompts should remind the user of the sentinel.

-**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 needed.

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. (tag: single entry, single exit)

Connecting the exit point of one control statement to the entry point of the next is called control statement stacking. A control statement appearing inside another is called control statement nesting.

Thus, algorithms in Java programs are constructed from only three kinds of control statements, combined in only two ways. This is the essence of simplicity. [UML Notes.docx](file:///C:\Computer%20Engineering\13-%20Software%20Engineering,%20Project%20Development\Software%20Engineering\UML\UML%20Notes.docx#structured_programming_summary)

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

-**if:** [selection\_statements](#selection_statements)

**-if else:** [selection\_statements](#selection_statements) Make sure you test for all cases.  
  
 **-Dangling else problem:** Java compiler associates an else with the immediately preceding if unless told to do otherwise by the placement of braces. The else below is associated with the second if. To make it associated with the first if, we need to put second if inside braces of first if.

if (x > 5)

if (y > 5)

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

else

System.out.println("x is <= 5");

**-Blocks:** The control statements expect one statement in their bodies. To include several statements in the body of a control statement, enclose the statements in braces. 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 an if else statement. The conditional operator is the only ternary operator in Java.** Together, the operands and the ?: symbol form a conditional expression. Use expressions of the same type for second and third operands of the conditional operator to avoid errors.

result = (point >= 50) ? "Successfull" : "Unsuccessfull";  
  
 System.out.printf("%s\n", (count % 2) == 1 ? "\*\*\*\*" : "++++++++");  
  
 **-switch:** [selection\_statements](#selection_statements)The switch statement consists of a block that contains a sequence of case labelsand an optional default case. If no match occurs between the controlling expression’s value and a case label, the default case executes. You should place the default case last.

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.

-Performance: The difference is not big for just a 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.

-Expressions in cases of switch: Each case must contain a constant integral expression (including enum constants) or a String. The expression in each case can also be a constant variable.

**-while** is generally used for sentinel controlled repetition. [repetition\_statements](#repetition_statements)

-Counter controlled while statement

int count = 1;

while (count <= 10) {

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

count++;

}

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.

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.

**-for** is generally used for counter controlled repetition. Contains elements of counter controlled repetition in one line called the for statement header. Initialization is done only once at the beginning. The increment happens after the body is executed, so pre and post increment are the same. counter++ is preferred since it is concise and increment happens after the body executes. Then the continuation test is performed again. [repetition\_statements](#repetition_statements)

-Multiple variables/conditions in for header: You can separate initializations and increments. But you can’t separate conditions. You have to merge the simple conditions into a complex condition.

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

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

-Sentinel controlled for statement

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 loop continuation condition is omitted, Java assumes that it is always true. There can still be one in the loop’s body.

You might omit the initialization expression if the program initializes the counter 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.

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

-**do while** [repetition\_statements](#repetition_statements)

do {

...  
 } while (count < 10);

-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.

**-Unconditional control statements**

**-break** statement causes immediate exit from repetition and switch statements.

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

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. Could use "count <= 10 && count != 5" instead of break.

-Label break: Used to exit out of more than one nested loop. The flow of control is not transferred to the label. The label just shows which repetition statement we broke from.   
 Label break only allows us to break out of one of the nested repetition statements we are in now.

first:

for (int i = 0; i < 2; i++)

second:

for (int j = 0; j < 3; j++)

break first; // "break third3;" won’t compile

**-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++) {

if (count == 5) // Could use "(count != 5)" instead of using continue

continue;

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

}

-Label continue

first:

for (int i = 0; i < 2; i++)

second:

for (int j = 0; j < 3; j++)

if (j == 1)

continue first;

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

**-return:** We can assign to a variable instead of using multiple return statements.

* **Methods: A Deeper Look**

The method header is the first line of a method where you specify the method’s return type, name, and parameter list (parameter types and names).

**-Program Modules in Java:** 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.   
 Don’t reinvent the wheel. When possible, reuse Java API classes and methods. This reduces program development time and avoids introducing programming errors.

Classes and methods help you modularize (divide-and-conquer) a program by separating its tasks into self-contained units. The statements in the method bodies are written only once (avoid repeating code), are hidden from other methods, and can be reused (software reusability) from several locations in a program.

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.

**-Why is main method static:** [static\_part1](#static_part1) 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. [command\_line\_arguments](#command_line_arguments)  
  
 **-main method** must be defined as “public static void main(String[] args)”, otherwise the JVM will not execute the application. You can overload the main method and call them from the default main method.  
 The program terminates when it reaches the end of the main method.

A class doesn’t have to have a main method. But it needs a main method to be executable by itself. When we have a class without a main method, we need another class that has a main method to use this class from. Those classes are called driver classes. (tag: client of a class)   
 We can have multiple classes in a .java file. The main method can be in all of them. Each class has its own .class file. So the main method of the .class file being executed is used.

The main method can have a return type. It can return int by using “System.exit(int);” or “return int;” when we are at the main method. The operating system interprets return value. 0 for succesful exit, 1 for unsuccesful exit. [System.exit(int)](https://stackoverflow.com/questions/2434592/difference-in-system-exit0-system-exit-1-system-exit1-in-java)

**-Final**

-Final variables: Any field declared with keyword final is *constant*—its value cannot change after the field is initialized. Final fields must be initialized in its declaration or in every constructor.

-Final reference variables prevent you from reassigning another object to them. But you can use methods and variables, and change the variables of these objects.

final A ob = new A();

ob.setI(6) // Works

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 classescannot be extended to create a subclass. 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.

**-Notes on Declaring and Using Methods:** 1) Using methodName() to call a static method of the same class, or an instance method of the same class from an instance method. 2) Using ClassName.methodName() to call a static method of a class. 3) Using objectName.methodName() to call and instance method of another class, or of the same class from static context.

-Returning a method: Reaching right brace. “return;”. “return value;”.

**-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 order so that each method can return to its caller.

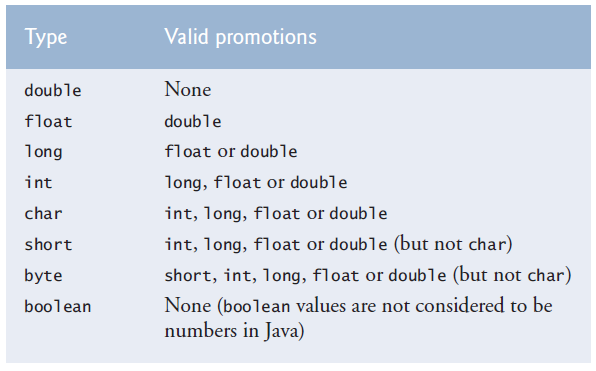
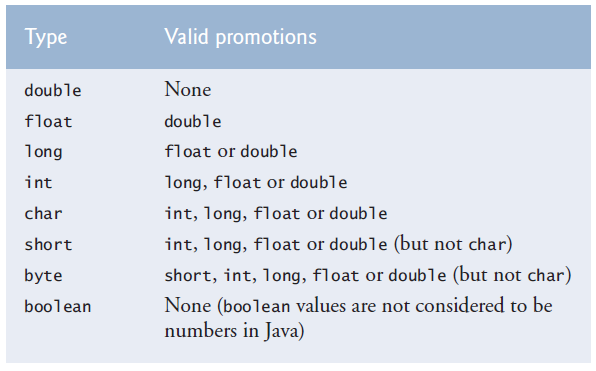
The method-call stack also contains the memory for the local variables (including the method parameters) used in each invocation of a method during a program’s execution. This data, stored as a portion of the method-call stack, is known as the stack frame (or activation record) of the method call. When 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.

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

**-Argument Promotion and Casting:** [data\_type\_conversions](#data_type_conversions)The promotion applies 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 allows 32bit float to hold the 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. SecureRandom objects produce nondeterministic random numbers that cannot be predicted. [Lineral congruential generator](https://en.wikipedia.org/wiki/Linear_congruential_generator) [Cryptographically secure pseudorandom number generator](https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator)

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.

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. You can omit all 3, in which case it will be 0 + 1 \* randomNumbers.nextInt(), which is just randomNumbers.nextInt().

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.

**-An enum** **type** declares a set of constants that are represented by identifiers called enum constants. We use constant identifier conventions while naming them. But these are not variables, they are the values. [enum\_part2](#enum_part2)

Defining new enum types can’t be done in methods. Must be done in field.

private enum Status { CONTINUE, WON, LOST };

Status gameStatus;

You can’t assign strings to an enum variable. You have to assign one of its values to it.

- gameStatus = "Won"; // Wrong

gameStatus = Status.WON; // Correct

- while (gameStatus == Status.CONTINUE)

-Why some constants are not defined as enum constants:Using an enum becomes unpractical if you need to compare the enum value with something else or use it in a calculation, etc. You can do it with switch statements but it defeats the purpose of using enums which is making the program more readable. In these cases use final variables. Named constants can be declared using both enum types and final variables.

**-Scope of a declaration** is the portion of the program that can refer to the declared entity by its name.

**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.

**-Method Overloading:** Methods of the same name can be declared in the same class, as long as they have different signatures—this is called method overloading. When an overloaded method is called, the compiler selects the appropriate method by examining the signature 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 Methods:Method calls cannot be distinguished only by return type. Overloaded methods can have different return types if the methods have different parameter lists.

Return types of methods alone don’t tell us which method we should use because we can’t know the return type before executing the method with the input.

[overloading\_main\_method](#overloading_main_method)

[variable\_length\_argument\_lists](#variable_length_argument_lists)

* **Arrays and ArrayLists**

Array objects are data structures consisting of related data items of the same type. Arrays remain the same length once they are created. Variables in an array are called elements or components.

Elements can be primitive or reference types (including arrays).  
 Every array object knows its own length and stores it in a length instance variable.

**-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.

Elements of local and field arrays get default values of their data type when the array is created.

- int[] array; // Declaration of arr. ref. and creation of arr. can be done discretely or together.

array = new int[10];

- int[] array = new int[10];

- int[] array; /\* We can use variables instead of literals for array creation expression and array initializer. Same for multi-dimensional arrays. \*/

array = new int[] { 1, 10, -7 };

Notice how you need an array creation expression 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.

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. The first index is 0. An index must be non-negative and an int value or a value of a type that can be promoted to int.

-Arrays of objects

Card[] deck;  
 deck = new Card[52];

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

deck[count] = new Card(faces[count % 13], suits[count / 13]);

**-ArrayIndexOutOfBoundsException:** 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. [exception\_handling\_part2](#exception_handling_part2)

-Implicit toString method: The toString method of an object is called implicitly when the object is used where a string is expected. For this effect to occur, toString must be declared with the header

public String toString()

**-NullPointerException** occurs when you try to call a method on a null reference. Ensuring that references are not null before you use them to call methods prevents NullPointerExceptions.

**-Enhanced for statement** iterates through the elements of an array or a collection without using a counter, thus eliminating several error possibilities caused by for header elements.

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

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

- for (int arrayColumn : array) // one dimensional array  
 System.out.println(arrayColumn);  
  
- for (int[] arrayRow : array) { // two dimensional array  
 for (int arrayColumn : arrayRow)  
 System.out.println(arrayColumn);  
 System.out.println();  
 }  
  
**-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. 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 or a reference type array element if you pass their reference to the method. But you can’t modify an element of an array if you pass a primitive type element.

-Creating an array in an argument list

method(new String[] { "hello", "goodbye" });

**-Pass by Reference, Pass by Value**  
  
With pass by value (call by value), a copy of the argument’s (primitive or reference) value is passed to the called method. The method works exclusively with the copy. Assigning a new value to the method’s copy doesn’t affect the original variable’s value in the caller, unlike pass by reference.  
 With pass by reference (call by reference), the reference of the argument is passed to the method so the method will use the same variable, not an equal variable.  
 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 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.  
 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 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, thus achieving the same effect.

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

You have to specify the length of the first dimension.

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

- int[][] array = {{ 1, 2, 3 }, { 4, 5, 6 }}; // Nested array initializer.

- array [0][0] = 10;  
  
-Multi-Dimensional arrays with different lengths

- int[][] array = new int[2][];

array[0] = new int[2];

array[1] = new int[3];  
  
- int[][] array = {{ 1, 2 }, { 3, 4, 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();  
 }  
  
- [traversing\_with\_enhanced\_for](#traversing_with_enhanced_for)

-Traversing a certain row 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.  
 You can supply an array as an argument for this parameter.  
  
 test(s1, s2, s3, s4);

public static void test(String... strings) {  
 System.out.println(s[0]);  
  
-Difference between array parameter and varargs parameter: If we were to use and array instead of varargs as stringArray parameter in the test method, it would not compile. It would require   
  
 test(new String[] { s1, s2, s3, s4 });

**-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. (tag: main method argument)  
 Command-line arguments are separated by white space, not commas. If the data you want to enter includes spaces then use quotes around your string.  
  
 java InitArray 5 0 4 // Command line

public static void main(String args[]) {

int num1 = Integer.parseInt(args[0]);

**-Class Arrays** (java.util.Arrays):[Class Arrays](http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html) Most of these methods work with single dim. arrays. You can divide the multi-dimension arrays into single dimension arrays and use them.  
  
- array1.equals(array); // Compares references

- boolean bool1 = Arrays.equals(array1, array2); // Compares references and contents. Ctrl + LMB in IDE to see the things it checks.  
  
- Arrays.sort(array);

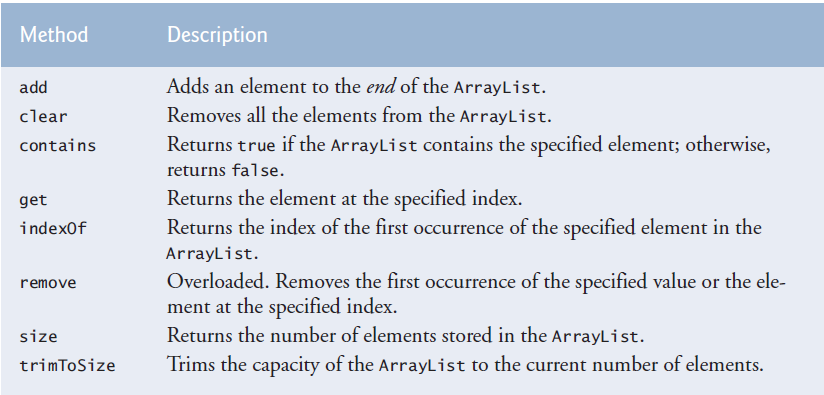
- Arrays.fill(array, 7);

- System.arrayCopy(src, srcPos, dest, destPost, length);

- int[] array1 = Arrays.copyOf(array2, length); // Return array with new length. Truncate or pad if necessary.

-int location =Arrays.binarySearch(array, 5); // Needs sorted array

- Arrays.toString(array); // use deepToString for nested arrays  
**-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. [collections\_part2](#collections_part2) [Class ArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)  
 ArrayList<T> can dynamically change its size to accommodate more elements. The T (by convention) is a placeholder—when declaring a new ArrayList, replace it with the type of elements that you want the ArrayList to hold.  
  
 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.  
  
 ArrayList<Integer> 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)  
  
  
  
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.

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

Contains method compares its argument to each element of the ArrayList in order, so using contains on a large Arraylist can be inefficient.

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>();

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**

**-Exception Handling with Time Class:** (Fig. 8.1) The class instance creation expression in the throw statement creates a new object of type IllegalArgumentException. 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. [exception\_handling\_part1](#exception_handling_part1) [exception\_handling\_part3](#exception_handling_part3)

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");

(Fig 8.2) Line 23 is 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 by the catch block, which 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 oriented. Clients are neither aware of, nor involved in, a class’s 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.

public Time2() {

this(0, 0, 0); // initializes each instance variable to zero

}

public Time2(int hour) {

this(hour, 0, 0); // hour supplied, minute and second defaulted to 0

}

public Time2(int hour, int minute) {

this(hour, minute, 0); // hour and minute supplied, second defaulted to 0

}

public Time2(int hour, int minute, int second) { // hour, minute and second supplied

if (hour < 0 || hour >= 24)

throw new IllegalArgumentException("hour must be 0-23");

if (minute < 0 || minute >= 60)

throw new IllegalArgumentException("minute must be 0-59");

if (second < 0 || second >= 60)

throw new IllegalArgumentException("second must be 0-59");

this.hour = hour;

this.minute = minute;

this.second = second;

}

public Time2(Time2 time) { // another Time2 object supplied

this(time.getHour(), time.getMinute(), time.getSecond());

}

public static void main(String[] args) {

Time2 t1 = new Time2(); // 00:00:00

Time2 t2 = new Time2(2); // 02:00:00

Time2 t3 = new Time2(21, 34); // 21:34:00

Time2 t4 = new Time2(12, 25, 42); // 12:25:42

Time2 t5 = new Time2(t4); // 12:25:42

}

When one object of a class has a 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 directly accessed private variables of the time object in the last constructor, and our object’s private variables in toUniversalTime and toString methods. But this is not recommended. When you use get and set methods, you just need to maintain methods that access the private variables directly.

**-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 Date birthDate;

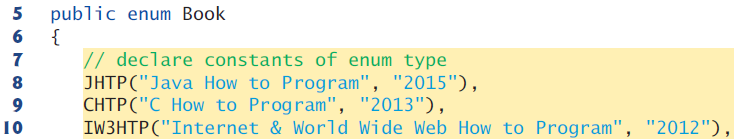
private Date hireDate;

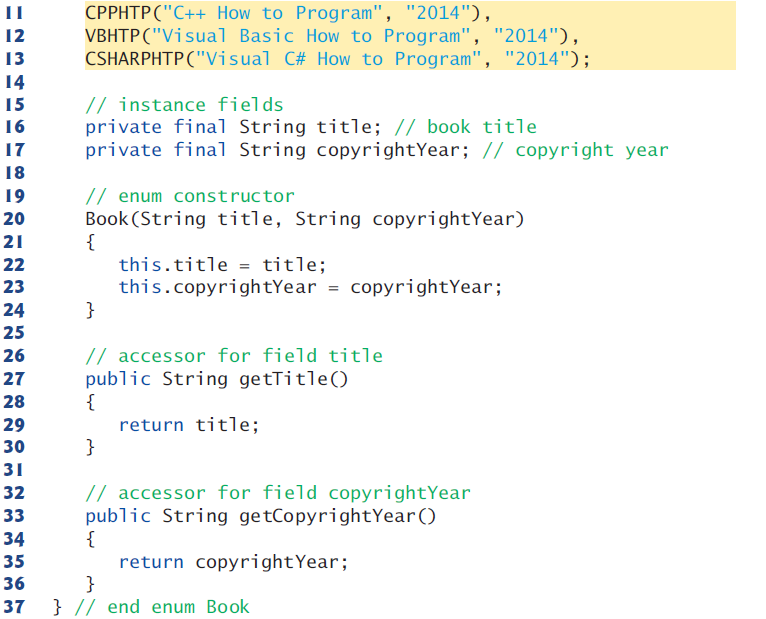
**-enum types** (java.util.EnumSet):[enum\_part1](#enum_part1) An enum type is declared with an enum declaration, which is a comma separated list of enum constants. The declaration may optionally include other components of traditional classes, such as constructors, fields, and methods.   
 Each enum declaration declares an enum class with the following restrictions. Enum constants are implicitly final and static. Any attempt to create an object of an enum type with operator new results in a compilation error.

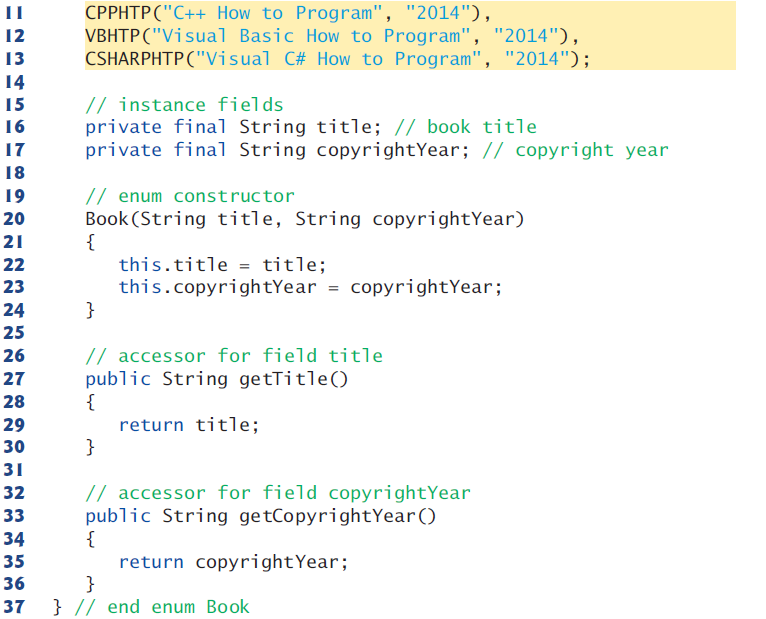
-Declaring Instance Variables, a Constructor, and Methods in an enum Type:Constructors of enums are called enum constructors.

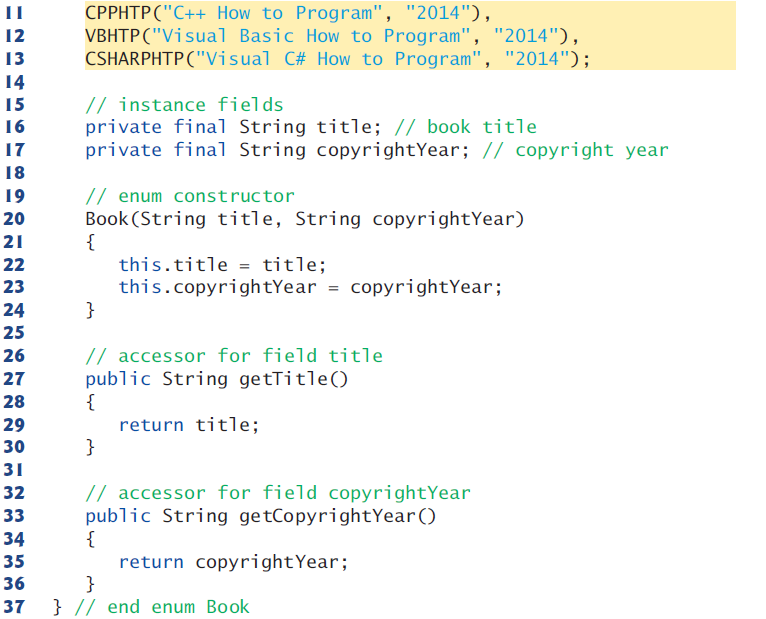
For every enum, the compiler generates the static method values that returns an array of the enum’s constants in the order they were declared.

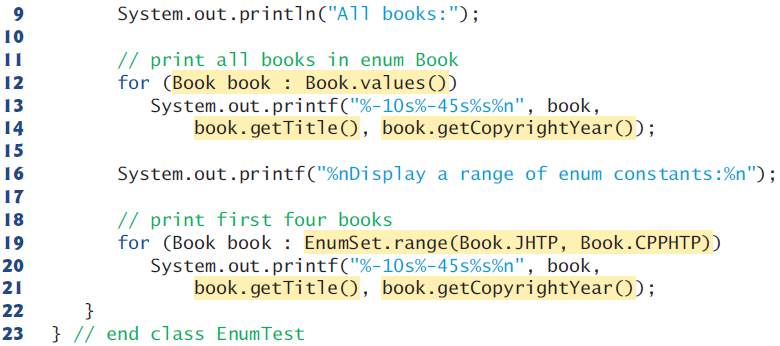
One of the methods of EnumSet is range. Which returns an EnumSet that contains all the constants between these two constants.

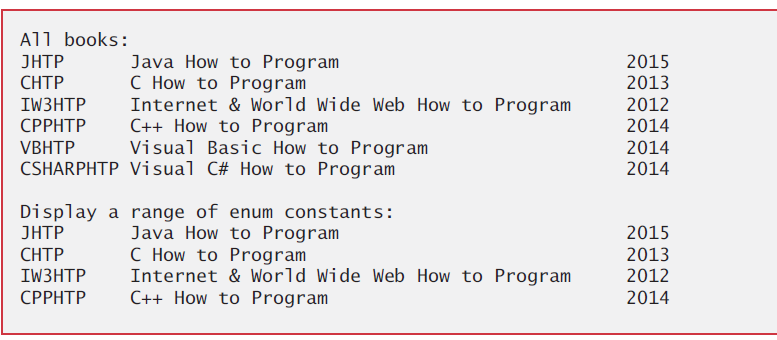












**-A static import** declaration enables you to import the static members of a class or interface.

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

import static java.lang.Math.sqrt;

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

import static java.lang.Math.\*;

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

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

* **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 realize many of the benefits 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 hierarchical structures. The class hierarchy is also called an inheritance hierarchy. Each arrow in the hierarchy represents an is-a relationship. [UML Notes.docx](file:///C:\Computer%20Engineering\13-%20Software%20Engineering,%20Project%20Development\Software%20Engineering\UML\UML%20Notes.docx#chapter9)  
 Objects of all classes that extend a common superclass can be treated as objects of that superclass.  
  
**-Members in inheritance:** Public and protected members are inherited.   
 Package private members are inherited if the subclass is in the same package as the superclass.  
 Private members are not inherited. They can be accessed through inherited methods.  
 The subclass can refer to inherited members 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 (.).  
  
[final\_methods](#final_methods) [static\_methods](#static_methods)  
  
**-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", "commission employee", firstName);

}

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 @Override annotation indicates 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. 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.

**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 doesn’t work because subclass can’t access private superclass instance variables.

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 is done 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.

[basePlusCommissionEmployee](#basePlusCommissionEmployee)

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

This version uses protected superclass instance variables.

-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  
  
This version uses private superclass instance 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

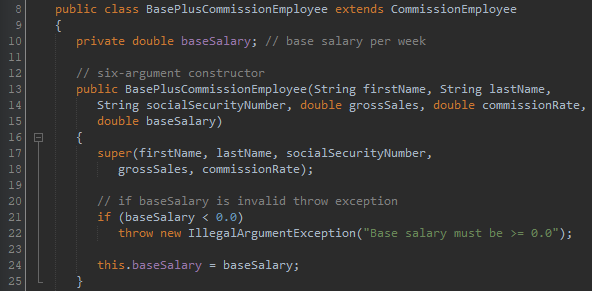
public String toString() {

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

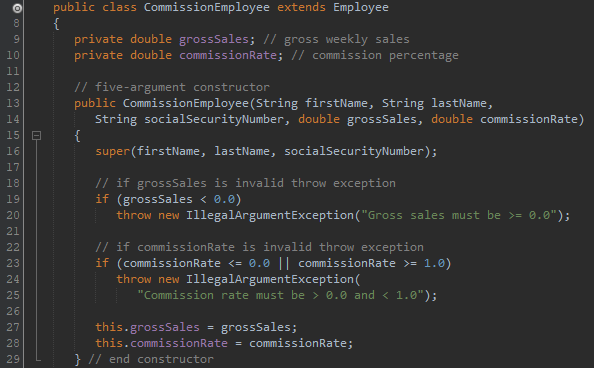
super.toString(), "base salary", getBaseSalary());

}

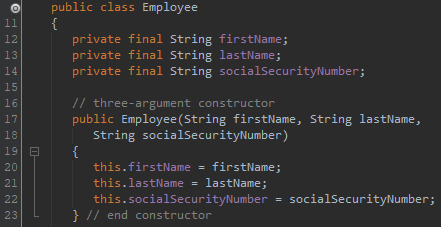
-Constructor parameter list and super argument list: In BasePlusCommissionEmployee constructor parameter list, we list all the parameters for CommissionEmployee and we also list the specialized parameters of BasePlusCommissionEmployee. The parameters for superclass are used for the superclass constructor call. The parameters for subclass are used in the subclass constructor.



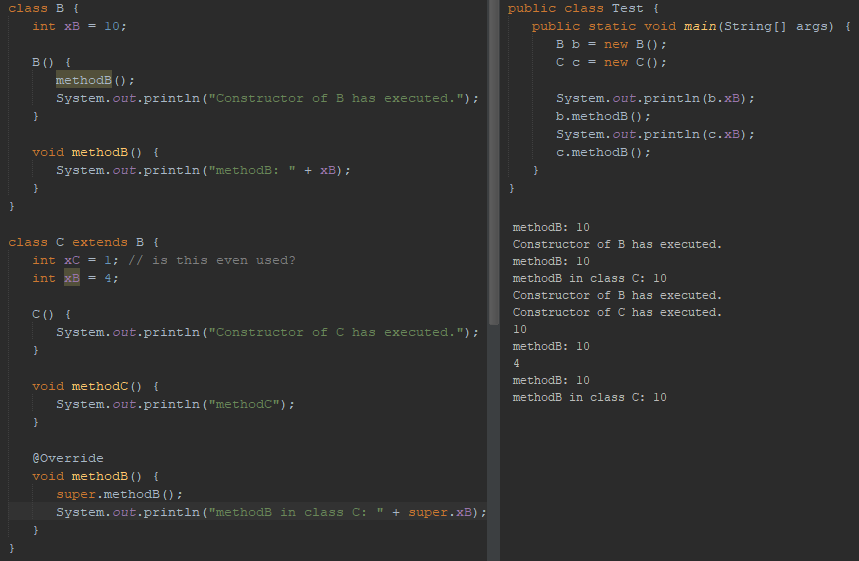
In CommissionEmployee constructor parameter list, we list all the parameters for Employee and we also list the specialized parameters of CommissionEmployee. The parameters for superclass are used for the superclass constructor call. The parameters for subclass are used in the constructor.

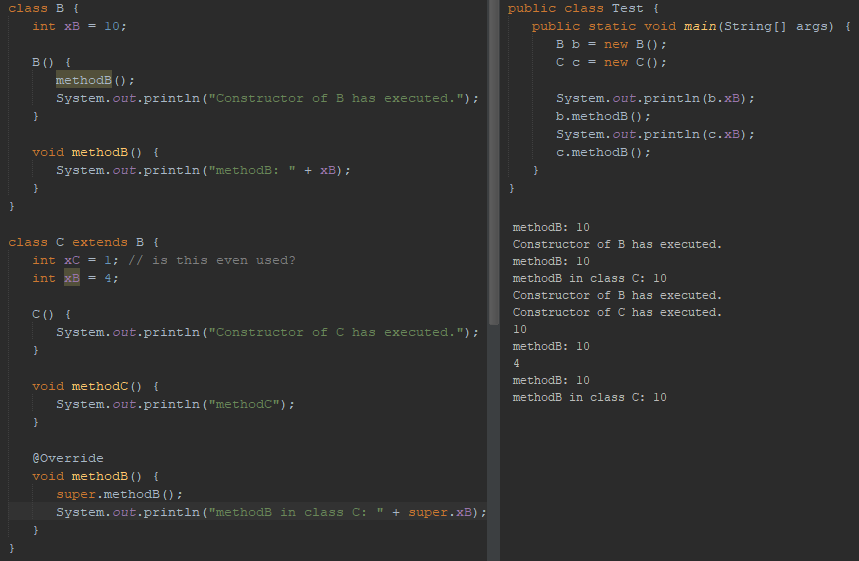


In Employee constructor parameter list, we list all the parameters of Employee.

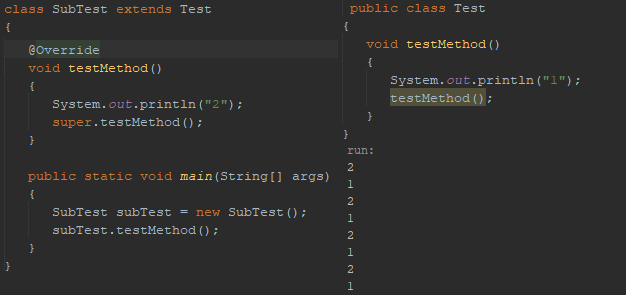


-Calling methods from superclass constructors/methods:When you create an object of a subclass, before the constructor of subclass executes, the constructor of superclass executes. And if there is a method call in superclass constructor, methods in subclass shadow methods in the superclass (if subclass has overridden 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 a method of superclass from subclass and if there is another method call in superclass method, methods in subclass shadow methods in the superclass (if subclass has overridden methods).



**-Class Object:** 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.

The methods of Object class are equals, hashCode, toString, wait, notify, notifyAll, getClass, finalize, and clone. (tags: shallow copy, deep copy) ([PDF](../2-%20Sources/-%20Java%20How%20to%20Program/Java%20How%20to%20Program,%2010th%20Edition.pdf) page 387)

* **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 superclass 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 binding, late binding, dynamic method dispatch, virtual method invocation) [static\_binding](#static_binding) [static\_binding2](#static_binding2)

**-Demonstrating Polymorphic Behavior:** Invoking a method on a subclass object via a superclass reference invokes the subclass functionality.

A a = new B(); // Casting subclass object to a superclass reference. Implicit upcasting

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.

a.brandNewMethodInB(); // illegal

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) does have an is-a relationship with class (B). Although it’s allowed, you should generally avoid downcasting.

B b = (B) a; // Casting superclass ref to a subclass var. Explicit downcasting

b.brandNewMethodInB(); // legal

If you invoke a static method, reference’s class’ will be called, not the object’s class’ because of static binding. This is the difference between overriding and hiding.

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 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 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 the object referenced by the lvalue is an rvalue. This would also be true for any object of an 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, 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 acceptable 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 don't 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 directly inherit only one class but classes and interfaces can implement multiple interfaces.

All objects of a class (and objects of its subclasses) that implements multiple interfaces have the is-a relationship with each implemented interface type.

public class ClassName extends SuperClassName implements FirstInterface,

SecondInterface, …

public abstract class Employee implements Payable

public class SalariedEmployee extends Employee

@Override // implement interface Payable method

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);

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. Comparable in Ch16 and 20, Serializable in Ch15 and 28, Runnable in Ch23, Event-listener (tag: event listener) in Ch12 and 22, and AutoCloseable in Ch11.

**-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 (ActionListener, Comparator, and Runnable).

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

-Encapsulation: Packaging related data and methods. Not sharing hidden data with outside directly. Using methods to access hidden data.

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

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

-Polymorphism: Allows us to use subclass objects 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-responsibility 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. This means we can extend this method to use new subclasses without modifying it.

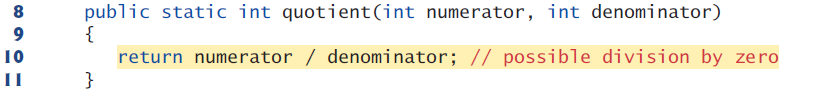
-Liskov substitution principle: Every subclass should be substitutable for its superclass.

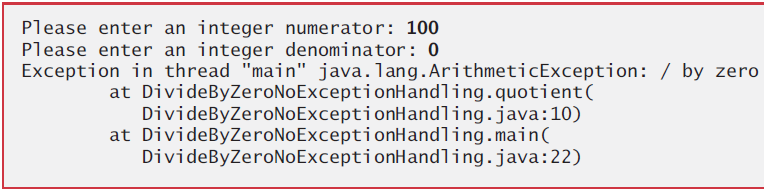
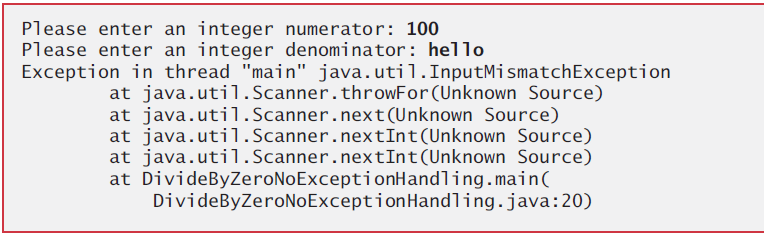
-Interface segregation principle: A client should never be forced to implement an interface that it doesn’t use or clients shouldn’t be forced to depend on methods they do not use.

-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.



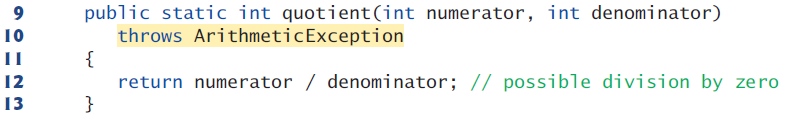
  
  


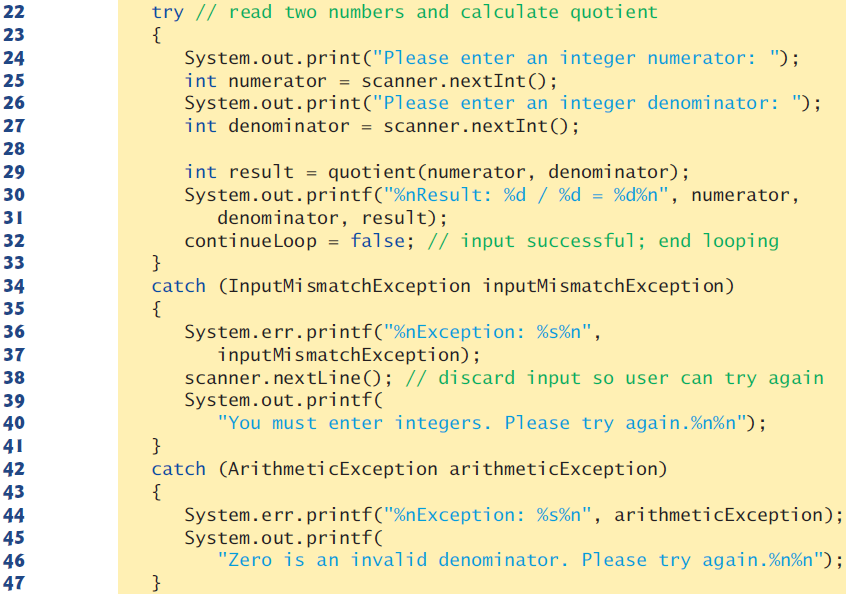
-Stack Trace: Several lines of information are displayed in response to invalid input. This information is known as a stack trace, which includes the name of the exception (java.lang.ArithmeticException) in a descriptive message that indicates the problem that occurred and the method-call stack (i.e., 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.

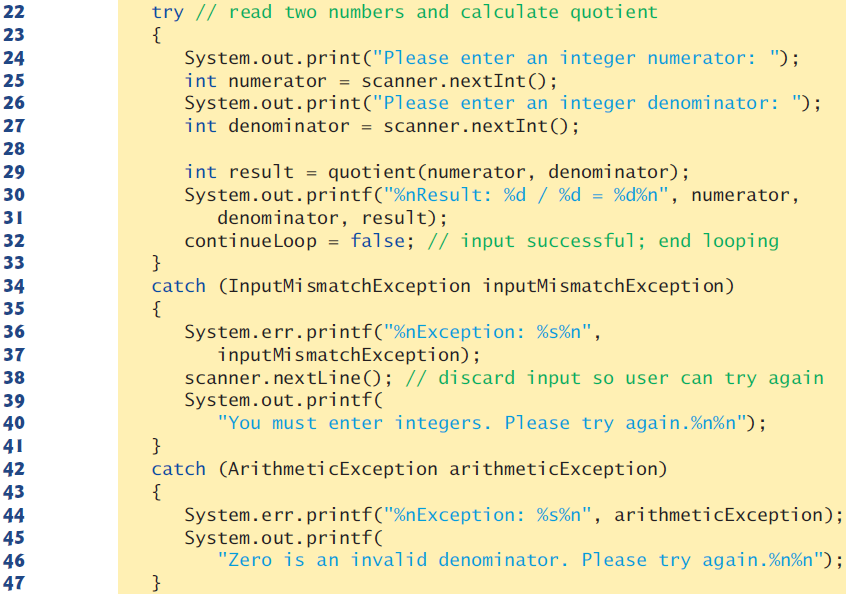
Each line of the stack trace contains the class name and method (e.g., DivideByZeroNoExceptionHandling.main) followed by the filename and line number (e.g., DivideByZeroNoExceptionHadling.java:22).

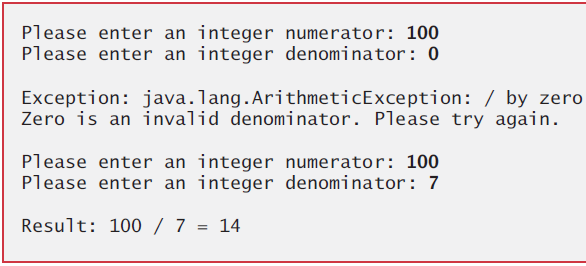
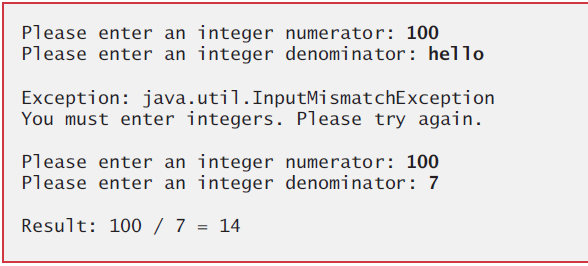
The top row of the call chain indicates the throw point—the initial point at which the exception occurred.

-Program Termination: 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 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 (standard error stream) object to output error messages.

System.out and System.err are streams—sequences of bytes. 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)

-Multi-catch: 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)

-Uncaught Exceptions: 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.

-Termination Model of Exception Handling: 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.

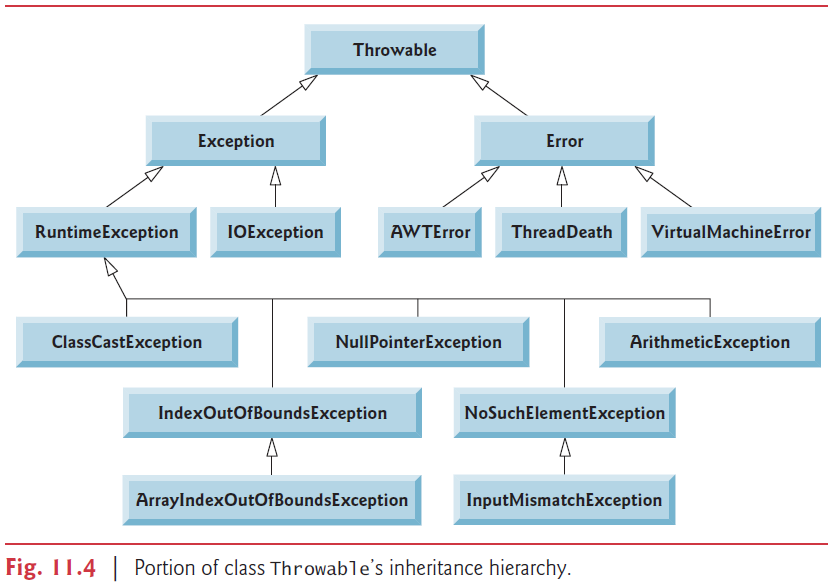
Read the online API documentation for a method before using it in a program. The documentation specifies the exceptions thrown by the method (if any) and indicates reasons why such exceptions may occur. Next, read the online API documentation for the specified exception classes. The documentation for 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).

**-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.

**-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)

-Unchecked exceptions: 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.

Provide exception-handling when proper coding can’t prevent unchecked exceptions such as NumberFormatException from Integer method parseInt.

-Checked Exceptions: 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.

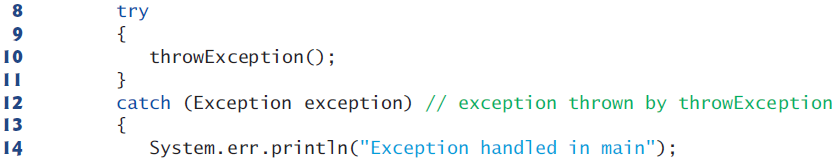
If a subclass method overrides a superclass method, it’s an error for the subclass method to list more exceptions in its throws clause than the superclass method does. However, a subclass’s throws clause can contain a subset of a superclass’s throws clause.

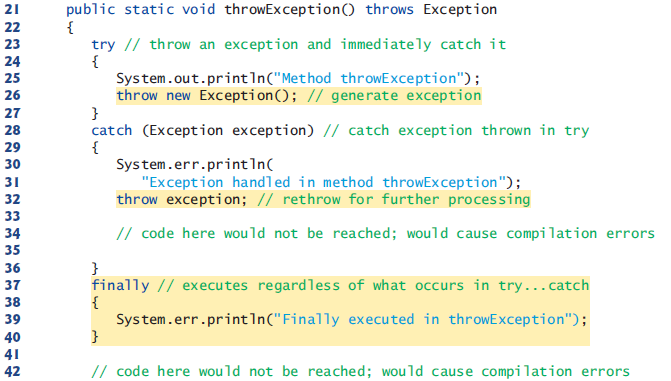
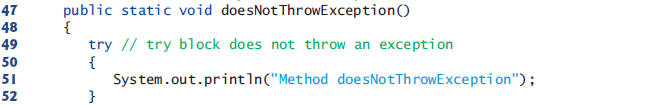
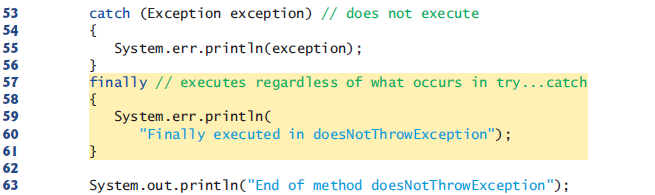
If your method calls other methods that throw checked exceptions, those exceptions must be caught or declared. If an exception can be handled meaningfully in a method, the method should catch the exception rather than declare it.  
  
**-Finally block (finally clause):** The finally block will execute 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.

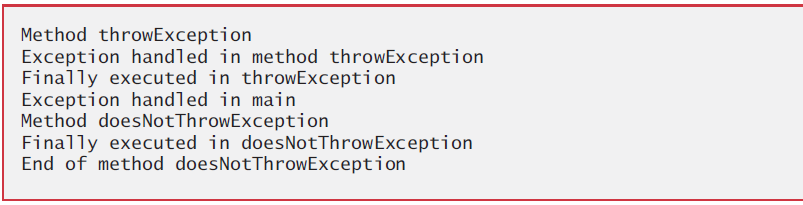
If an exception that occurs in a try block cannot be caught by one of that try block’s catch handlers, the program skips the rest of the try block and control proceeds to the finally block. Then the program passes the exception to the next outer try block—normally in the calling method—where an associated 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 latest 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.





-The throw Statement: You can throw exceptions yourself by using the throw statement.

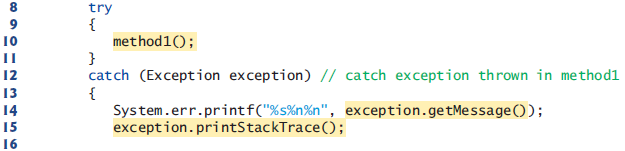
Throw exceptions from constructors to indicate that the constructor parameters are not valid—this prevents an object from being created in an invalid state.

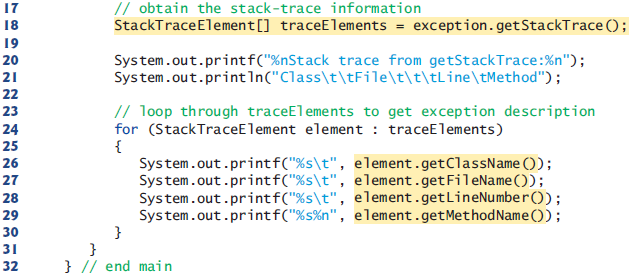
-Rethrowing Exceptions is done when a catch block, upon receiving an exception, decides either that it cannot process that exception or that it can only partially process it.

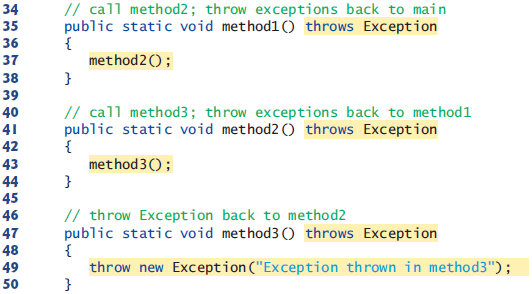
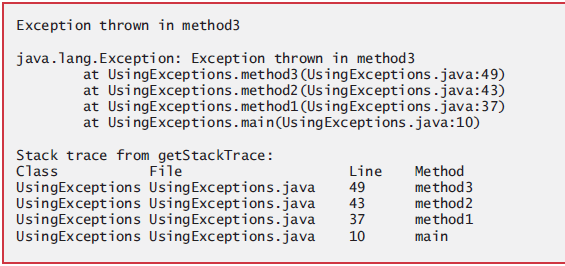
Avoid placing in a finally block code that can throw an exception. If such code is required,

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

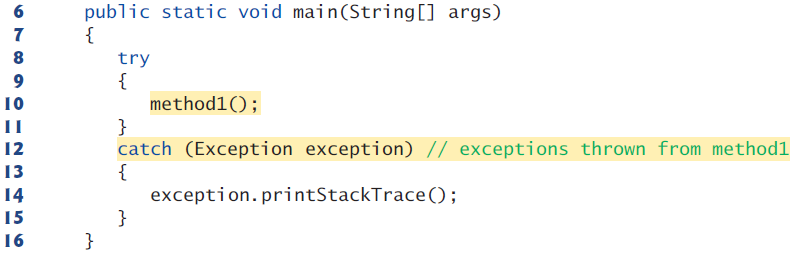
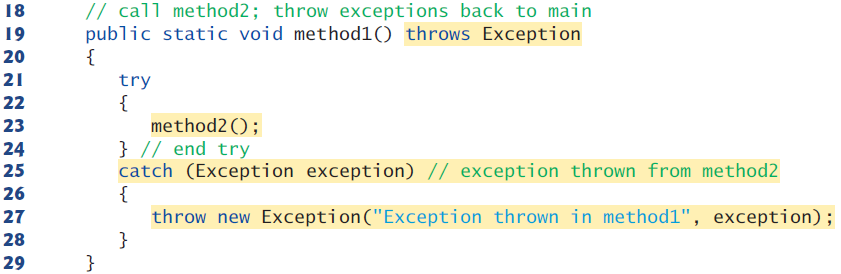
**-Stack Unwinding:** When an exception is thrown but not caught in a particular scope, the method-call stack is “unwound,” and an attempt is made to catch the exception in the next outer try block. This process is called stack unwinding. Unwinding the method-call stack means that the method in which 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.

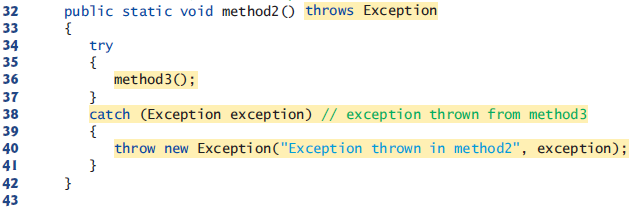
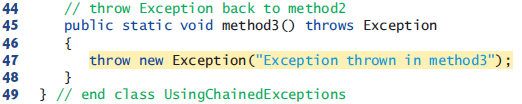
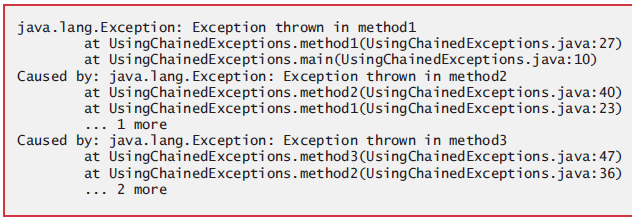




-Obtaining Information from an Exception Object: The getMessage method returns the descriptive string stored in an exception. The toString method returns a String containing the name of the exception’s class and a descriptive message. The printStackTrace method outputs the stack trace to the standard error stream. The getStackTrace method retrieves the stack trace.  
 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. 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) Takes no arguments and passes a default error message String to the superclass constructor

2) Receives a customized error message as a String and passes it to the superclass constructor

3) Receives a customized error message as a String and a Throwable (for chaining exceptions) and passes both to the superclass constructor.

4) 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:** 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.

When their preconditions and postconditions are not met, methods typically throw exceptions.  
 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. 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 throws an AssertionError with expression2 as the error message if expression1 is false.

Use assertions only during development, not in production code. 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

java -ea AssertTest

**-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. The try-with-resources statement implicitly calls the theObject’s close method at the end of the try block. (tag: try with resources)

You can allocate multiple resources by separating them with a semicolon (;).

try (*ClassName* theObject = new *ClassName*()) {

// use theObject here

}

catch (Exception e) {

// catch exceptions that occur while using the resource

}

* **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 a handler for. To remove a handler, properties window (below palette), Events tab.  
  
**-Simple GUI-Based Input/Output with JOptionPane** (javax.swing.JoptionPane): GUI applications are made of windows and dialog boxes (also called dialogs). There are message, input, confirm, and option 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)in which the user can enter text. Clicking OK or pressing Enter returns the string. Clicking Cancel or pressing Esc returns null.  
 Inputting a non string value can be done in one line.  
  
 int age = Integer.parseInt(JOptionPane.showInputDialog("Enter your age"));  
  
Third argument is title. Fourth argument is messageType. JOptionPane.PLAIN\_MESSAGE , ERROR\_MESSAGE, INFORMATION\_MESSAGE, WARNING\_MESSAGE, QUESTION\_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.  
­  
**-Overview of Swing Components:** AWT components look like the native GUI components of the platform. Swing 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.  
 Both text fields and text areas have the getText, getSelectedText, and setText methods.  
  
 // Pressing enter in studentNumber text field puts focus on password field  
 password.requestFocus();  
 // Pressing enter in password field invokes manualSubmit button  
 manualSubmitActionPerformed(evt);   
 // Pressing enter in textField1 sets textField2   
 textField2.setText(String.format("textField1: %s", textField1.getText()));  
  
-JTextArea:Like multiple-selection JLists, code the logic to 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 a String object.  
  
**-Nested Classes:** 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. [nested\_class\_part1](#nested_class_part1)  
 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.  
 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.  
  
 imagesComboBox.addItemListener(new ItemListener() {  
 ...  
 });  
  
**-Adapter Classes:** For many event-listeners that contain multiple methods, you don’t 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 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);  
 textField.setFont(font); **-JComboBox:** JComboBoxis a generic class. It displays a String representation of each object. You can hard code values using the model property. Or, right click, customize code,

jComboBox1 = new javax.swing.JComboBox<String>(names);  
 Some useful methods for combo boxes are getItemCount, addItem, getSelectedItem. We can use getClass().getResource() to get the URL of a resource. ([PDF](../2-%20Sources/-%20Java%20How%20to%20Program/Java%20How%20to%20Program,%2010th%20Edition.pdf) page 505)  
  
**-JList:** JListis a generic class.It supports both single and multiple selection lists. You can hard code values using the model property. Or, right click, customize code,  
  
 jList1 = new javax.swing.JList<String>(names);

When we are using a single selection list, we code the logic into the list (listSelection event). But when we use a multiple selection list, we code the logic into another component such as a button.  
 Some useful methods for lists are getSelectedIndex and getSelectedValue.  
  
 copyList.setListData(colorList.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.

**-mouseDragged**

currentLocation = label.getLocation(); // In mousePressed method

offsetAmount = evt.getPoint();

changeAmount = evt.getPoint(); // In mouseDragged method

currentLocation.setLocation(currentLocation.x + changeAmount.x - offsetAmount.x,

currentLocation.y + changeAmount.y - offsetAmount.y);

label.setLocation(currentLocation);

**-Key Event Handling:** First code is used by pressed and released for nonprintable chars. KeyEvent.getKeyText(evt.getKeyCode())  
  
 evt.getKeyChar() // used by all 3 methods for printable characters

**-MenuBar:** Use “Alt + Key” and “Key” for mnemonics of menus and menu items respectively. Menu item accelerators/shortcuts can be used without opening the menu.

**-Popup Menu:** Drag a popup menu and menu items to Design. Move menu items to popup menu in Navigator. Then create a MouseClicked method for a component.

if (evt.isMetaDown())

popupMenu.show(panel, evt.getX(), evt.getY());

**-File chooser and color chooser:**  Drag file and color choosers outside borders of your GUI window to put them in “Other components”. Code a button.

- fileChooser.setCurrentDirectory(new File("C:\\Users\\Michael\\Desktop"));

int response = fileChooser.showOpenDialog(rootPane); // showSaveDialog

if (response == fileChooser.APPROVE\_OPTION)

File file = new File(fileChooser.getSelectedFile().getAbsolutePath());

- Color color = colorChooser.showDialog(rootPane, "title", Color.BLACK);

**-Open a new GUI window**:  
  
 this.dispose(); // frame.dispose();  
 JFrame2.main(null);

* **Strings, Characters and Regular Expressions**

**-Class String:** To conserve memory, Java treats all string literals with the same contents as a single String object in string pool that has many references to it.

String objects are immutable—they cannot be modified after they are created. [Class String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html)  
  
 **-equals:** Put the value/variable you know for sure is a string to the left to prevent exceptions.  
 == operator checks whether two objects are same (same reference), not whether two objects are equal (same content).

**-Scanner, StringTokenizer, split:** [split](#split)

- Scanner fromString1 = new Scanner(string1);

while (fromString1.hasNext()) {  
 System.out.println(fromString1.next());

- StringTokenizer st = new StringTokenizer(string1);

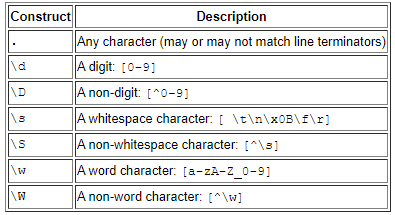
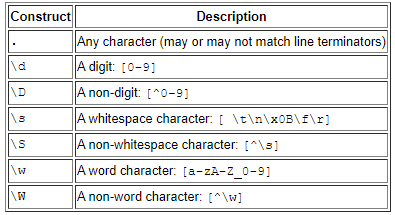
while (st.hasMoreTokens())

System.out.println(st.nextToken());

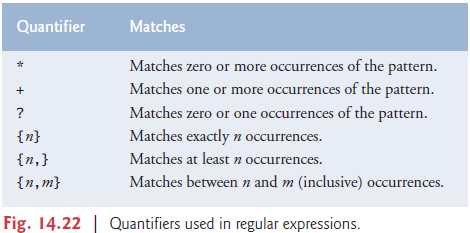
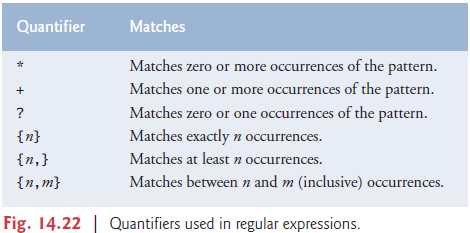
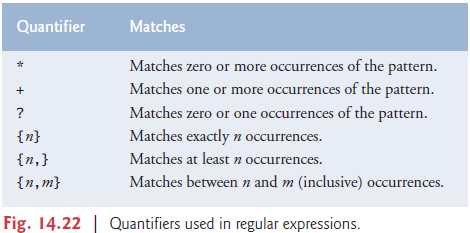
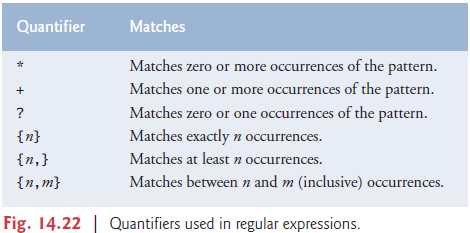
**-String.format()**

- String string1 = String.format("No: %s String: \"%s\"", 3 \* x, string2);

- String string1 = "No: " + 3 \* x + " String: \"" + string2 + "\"";

**-Class StringBuilder** provides modifiable strings which are often more efficient in programs that frequently perform string concatenation or other string modifications. [Class StringBuilder](https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html)  
  
 **-StringBuilder Methods length, capacity, setLength, and ensureCapacity:** If a StringBuilder is going to increase greatly in size, possibly multiple times, setting its capacity high at the beginning will increase performance.  
  
**-Class Character:** [Class Character](https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html) **-Regular Expressions, Class Pattern and Class Matcher:** A regular expression is a String that describes a search pattern for matching characters in other Strings.  
  
****

To match a set of characters that does not have a predefined character class, use square brackets, [].  
 Character ranges are represented by placing a dash (-) between two characters.  
 If the first character in the brackets is "^", the expression accepts any character other than those indicated.  
 "|" matches the expression to its left or to its right.  
 “\” escapes a special character: . \* + ? $ ^ / \ [ ] { } ( ). The Java String "\\\*" represents the regular-expression pattern \\* which matches a single \* character in the search string.

All of the quantifiers are greedy. This means that they’ll match as many occurrences as they can as long as the match is still successful. However, if any of these quantifiers is followed by a question mark (?), the quantifier becomes reluctant (sometimes called lazy). It then will match as few occurrences as possible as long as the match is still successful.  
  
  
  
The first arguments of matches, replaceAll, replaceFirst, and split methods are regular expressions. Both arguments of replace method are either strings or chars.  
  
 return lastName.matches("[a-zA-Z]+(['-][a-zA-Z]+)\*");   
 return address.matches("\\d+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)");  
 return phone.matches("[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}");  
  
 firstString = firstString.replaceAll("stars", "carets");  
 secondString = secondString.replaceFirst("\\d", "digit");  
 String[] results = secondString.split(",\\s\*"); // split(" ")

If a regular expression will be used more than once, it’s more efficient to create a Pattern object. Method find attempts to match a piece of the search object to the search pattern. Each call to this method starts at the point where the last call ended, unlike method lookingAt.  
  
 Pattern pattern = Pattern.compile("J.\*\\d[0-35-9]);

String string1 = "Jane's 05\nDave's 11\nJohn's 04\nJoe's 12";

Matcher matcher = pattern.matcher(string1);

while (matcher.find())

System.out.println(matcher.group());

* **Advanced I/O and Serialization** (java.io.\*) [streams\_part3](#streams_part3)

- Formatter output = new Formatter("C:\\src\\textfile.txt"); // open to write  
  
 while (input.hasNext()) // standard input object  
 output.format("%s%n", input.next()); // write  
 output.close();  
  
- Scanner input = new Scanner(Paths.get("clients.txt")); // open to read  
  
 while (input.hasNext())  
 System.out.printf("%-10d%-12s%n", input.nextInt(), input.next()); // read  
 input.close();

**-Serialize:** The class of the object needs to implement serializable. Transient fields aren’t serialized.

- FileOutputStream fileOut = new FileOutputStream("UserInfo.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut);

out.writeObject(user); // out.close(); fileout.close();

- FileInputStream fileIn = new FileInputStream("C:\\UserInfo.ser");

ObjectInputStream in = new ObjectInputStream(fileIn);

User user = (User) in.readObject(); // in.close(); fileIn.close();

* **Collections** [collections\_part1](#collections_part1)