COOL CRAM BIBLE

For Java Certification Course 310-025

Sun Certified Java Programmer

Welcome to Visit our Web Site

www.coolcram.com

Feel free to contact us via

Email:support@coolcram.com

Cool Cram Lab

Sun Certified Java Programmer Certification JAVA 2 Study Notes

Exam Objectives

Section 1 Declarations and Access Control

- Write code that declares, constructs, and initializes arrays of any base type using any of the permitted forms both for declaration and for initialization.
- Declare classes, inner classes, methods, instance variables, static variables, and automatic (method local) variables making appropriate use of all permitted modifiers (such as public, final,static, abstract, and so forth). State the significance of each of these modifiers both singly and in combination, and state the effect of package relationships on declared items qualified by these modifiers.
- For a given class, determine if a default constructor will be created, and if so, state the prototype of that constructor.
- State the legal return types for any method given the declarations of all related methods in this or parent classes.

Section 2 FLOW CONTROL AND EXCEPTION HANDLING

- Write code using if and switch statements and identify legal argument types for these statements.
- Write code using all forms of loops including labeled and unlabeled use of break and continue, and state the values taken by loop control variables during and after loop execution.
- Write code that makes proper use of exceptions and exception handling clauses (try, catch, finally) and declares methods and overriding methods that throw exceptions.

Section 3 Garbage Collection

- State the behavior that is guaranteed by the garbage collection system, and write code that explicitly makes objects eligible for collection.

Section 4 Language Fundamentals

- Identify correctly constructed source files, package declarations, import statements, class declarations (of all forms including inner classes), interface declarations and implementations (for java.lang.Runnable or other interface described in the test), method declarations (including the main method that is used to start execution of a class), variable declarations and identifiers.
- State the correspondence between index values in the argument array passed to a main method and command line arguments. Identify all Java Programming Language keywords and correctly constructed identifiers.
- State the effect of using a variable or array element of any kind when no explicit assignment has been made to it.
- State the range of all primitive data types and declare literal values for String and all primitive types using all permitted formats, bases, and representations.

Section 5 Operators and assignments

- Determine the result of applying any operator, including assignment operators, instanceof, and casts to operands of any type, class, scope, or accessibility, or any combination of these.
- Determine the result of applying the boolean equals(Object) method to objects of any combination of the classes java.lang.String, java.lang.Boolean, and java.lang.Object.
- In an expression involving the operators &, |, &&, ||, and variables of known values state which operands are evaluated and the value of the expression.
- Determine the effect upon objects and primitive values of passing variables into methods and performing assignments or other modifying operations in that method.

Section 6 Overloading, Overriding, Runtime Type, and Object Orientation

- State the benefits of encapsulation in object oriented design and write code that implements tightly encapsulated classes and the relationships "is a" and "has a".

- Write code to invoke overridden or overloaded methods and parental or overloaded constructors; and describe the effect of invoking these methods.
- Write code to construct instances of any concrete class including normal top level classes, inner classes, static inner classes, and anonymous inner classes.

Section 7 Threads

- Write code to define, instantiate, and start new threads using both java.lang.Thread and java.lang.Runnable.
- Recognize conditions that might prevent a thread from executing.
- Write code using synchronized, wait, notify, or notifyAll, to protect against concurrent access problems and to communicate between threads. Define the interaction between threads and between threads and object locks when executing synchronized, wait, notify, or notifyAll.

Section 8 The java.awt PACKAGE

- Write code using component, container, and LayoutManager classes of the java.awt package to present a GUI with specified appearance and resize behavior, and distinguish the responsibilities of layout managers from those of containers.
- Write code to implement listener classes and methods, and in listener methods, extract information from the event to determine the affected component, mouse position, nature, and time of the event. State the event classname for any specified event listener interface in the java.awt.event package.

Section 9 The java.lang PACKAGE

- Write code using the following methods of the java.lang.Math class: abs, ceil, floor, max, min, random, round, sin, cos, tan, sqrt.
- Describe the significance of the immutability of String objects.

Section 10 The java.util PACKAGE

- Make appropriate selection of collection classes/interfaces to suit specified behavior requirements.

Section 11 The java.io PACKAGE

- Write code that uses objects of the file class to navigate a file system.
- Write code that uses objects of the classes InputStreamReader and OutputStreamWriter to translate between Unicode and either platform default or ISO 8859-1 character encoding and Distinguish between conditions under which platform default encoding conversion should be used and conditions under which a specific conversion should be used.
- Select valid constructor arguments for FilterInputStream and FilterOutputStream subclasses from a list of classes in the java.io.package.
- Write appropriate code to read, write and update files using FileInputStream, FileOutputStream, and RandomAccessFile objects.
- Describe the permanent effects on the file system of constructing and using FileInputStream, FileOutputStream, and RandomAccessFile objects.

What are the potential trips/traps in SCJP exam?

- Two public classes in the same file.
- Main method calling a non-static method.
- Methods with the same name as the constructor(s).
- Thread initiation with classes that dont have a run() method.
- Local inner classes trying to access non-final vars.
- Case statements with values out of permissible range.
- Math class being an option for immutable classes !!
- instanceOf is not same as instanceof
- Private constructors
- An assignment statement which looks like a comparison if (a=true)...
- System.exit() in try-catch-finally blocks.
- Uninitialized variable references with no path of proper initialization.
- Order of try-catch-finally blocks matters.
- main() can be declared final.
- -0.0 == 0.0 is true.
- A class without abstract methods can still be declared abstract.
- RandomAccessFile descends from Object and implements DataInput and DataOutput.
- Map doesnot implement Collection.
- Dictionary is a class, not an interface.
- Collection is an Interface where as Collections is a helper class.
- Class declarations can come in any order (derived first, base next etc.).
- Forward references to variables gives compiler error.
- Multi dimensional arrays can be sparce ie., if you imagine the array as a matrix, every row need not have the same number of columns.
- Arrays, whether local or class-level, are always initialized,
- Strings are initialized to null, not empty string.
- An empty string is NOT the same as a null string.
- A declaration cannot be labelled.
- continue must be in a loop(for, do , while). It cannot appear in case constructs.
- Primitive array types can never be assigned to each other, eventhough the primitives themselves can be assigned. ie., ArrayofLongPrimitives = ArrayofIntegerPrimitives gives compiler error eventhough longvar = intvar is perfectly valid.
- A constructor can throw any exception.
- Initilializer blocks are executed in the order of declaration.
- Instance initializer(s) gets executed ONLY IF the objects are constructed.
- All comparisons involving NaN and a non-Nan would always result false.
- Default type of a numeric literal with a decimal point is double.
- integer (and long) operations / and % can throw ArithmeticException while float / and % will never, even in case of division by zero.
- == gives compiler error if the operands are cast-incompatible.
- You can never cast objects of sibling classes(sharing the same parent), even with an explicit cast.
- .equals returns false if the object types are different.It does not raise a compiler error.
- No inner class can have a static member.
- File class has NO methods to deal with the contents of the file.
- InputStream and OutputStream are abstract classes, while DataInput and DataOutput are interfaces.

Chapter 1 Language Fundamentals

1. Source file's elements (in order)
    a. Package declaration
    b. Import statements
    c. Class definitions
2. Importing packages doesn't recursively import sub-packages.
3. Sub-packages are really different packages, happen to live within an enclosing package. Classes in sub-packages cannot access classes in enclosing package with default access.
4. Comments can appear anywhere. Can't be nested.(No matter what type of comments)
5. At most one public class definition per file. This class name should match the file name. If there are more than one public class definitions, compiler will accept the class with the file's name and give an error at the line where the other class is defined.
6. It's not required having a public class definition in a file. Strange, but true. ☺ In this case, the file's name should be different from the names of classes and interfaces (not public obviously).
7. Even an empty file is a valid source file.
8. An identifier must begin with a letter, dollar sign ($) or underscore (_). Subsequent characters may be letters, $, _ or digits.
9. An identifier cannot have a name of a Java keyword. Embedded keywords are OK. true, false and null are literals (not keywords), but they can't be used as identifiers as well.
10. const and goto are reserved words, but not used.
11. Unicode characters can appear anywhere in the source code. The following code is valid.
```
ch\u0061r a = 'a';
char \u0062 = 'b';
char c = '\u0063';
```
12. Java has 8 primitive data types.

Data Type	Size (bits)	Initial Value	Min Value	Max Value
boolean	1	false	false	true
byte	8	0	-128 (-2^7)	127 ($2^7 - 1$)
short	16	0	-2^{15}	2^{15} - 1
char	16	'\u0000'	'\u0000' (0)	'\uFFFF' ($2^{16} - 1$)
int	32	0	-2^{31}	2^{31} - 1
long	64	0L	-2^{63}	2^{63} - 1
float	32	0.0F	1.4E-45	3.4028235E38
double	64	0.0	4.9E-324	1.7976931348623157E308

13. All numeric data types are signed. char is the only unsigned integral type.
14. Object reference variables are initialized to null.
15. Octal literals begin with zero. Hex literals begin with 0X or 0x.
16. Char literals are single quoted characters or unicode values (begin with \u).
17. A number is by default an int literal, a decimal number is by default a double literal.
18. 1E-5d is a valid double literal, E2d is not (since it starts with a letter, compiler thinks that it's an identifier)
19. Two types of variables.
    1. Member variables
        - Accessible anywhere in the class.
        - Automatically initialized before invoking any constructor.
        - Static variables are initialized at class load time.
        - Can have the same name as the class.
    2. Automatic variables(method local)
        - Must be initialized explicitly. (Or, compiler will catch it.) Object references can be initialized to null to make the compiler happy. The following code won't compile.
            ```
            public String testMethod ( int a ) {
                    String tmp;
            ```

if (a > 0) tmp = "Positive"; // Specify else part to make this code compile.
return tmp;
}

- Can have the same name as a member variable, resolution is based on scope.

20. Arrays are Java objects. If you create an array of 5 Strings, there will be 6 objects created.
21. Arrays should be
    1. Declared. (int[] a; String b[]; Object []c; Size should not be specified now)
    2. Allocated (constructed). (a = new int[10]; c = new String[arraysize])
    3. Initialized. for (int i = 0; i < a.length; a[i++] = 0)
22. The above three can be done in one step. int a[] = { 1, 2, 3 }; or int a[] = new int[] { 1, 2, 3 }; But never specify the size with the new statement.
23. Java arrays are static arrays. Size has to be specified at compile time. Array.length returns array's size. (Use Vectors for dynamic purposes).
24. Array size is never specified with the reference variable, it is always maintained with the array object. It is maintained in array.length, which is a final instance variable.
25. Anonymous arrays can be created and used like this: new int[] {1,2,3} or new int[10]
26. Arrays with zero elements can be created. args array to the main method will be a zero element array if no command parameters are specified. In this case args.length is 0.
27. Comma after the last initializer in array declaration is ignored.

        int[] i = new int[2] { 5, 10}; // Wrong
        int i[5] = { 1, 2, 3, 4, 5}; // Wrong
        int[] i[] = {{ }, new int[] {} }; // Correct
        int i[][] = { {1,2}, new int[2] }; // Correct
        int i[] = { 1, 2, 3, 4, } ; // Correct

28. Array indexes start with 0. Index is an int data type.
29. Square brackets can come after datatype or before/after variable name. White spaces are fine. Compiler just ignores them.
30. Arrays declared even as member variables also need to be allocated memory explicitly.
        static int a[];
        static int b[] = {1,2,3};
        public static void main(String s[]) {
                System.out.println(a[0]); // Throws a null pointer exception
                System.out.println(b[0]); // This code runs fine
                System.out.println(a); // Prints 'null'
                System.out.println(b); // Prints a string which is returned by toString
        }
31. Once declared and allocated (even for local arrays inside methods), array elements are automatically initialized to the default values.
32. If only declared (not constructed), member array variables default to null, but local array variables will not default to null.
33. Java doesn't support multidimensional arrays formally, but it supports arrays of arrays. From the specification - *"The number of bracket pairs indicates the depth of array nesting."* So this can perform as a multidimensional array. (no limit to levels of array nesting)
34. In order to be run by JVM, a class should have a main method with the following signature.
        public static void main(String args[])
        static public void main(String[] s)
35. args array's name is not important. args[0] is the first argument. args.length gives no. of arguments.
36. main method can be overloaded.
37. main method can be final.
38. A class with a different main signature or w/o main method will compile. But throws a runtime error.
39. A class without a main method can be run by JVM, if its ancestor class has a main method. (main is just a method and is inherited)
40. Primitives are passed by value.

41. Objects (references) are passed by reference. The object reference itself is passed by value. So, it can't be changed. But, the object can be changed via the reference.
42. Garbage collection is a mechanism for reclaiming memory from objects that are no longer in use, and making the memory available for new objects.
43. An object being no longer in use means that it can't be referenced by any 'active' part of the program.
44. Garbage collection runs in a low priority thread. It *may* kick in when memory is too low. No guarantee.
45. It's not possible to force garbage collection. Invoking System.gc *may* start garbage collection process.
46. The automatic garbage collection scheme guarantees that a reference to an object is always valid while the object is in use, i.e. the object will not be deleted leaving the reference "dangling".
47. There are no guarantees that the objects no longer in use will be garbage collected and their finalizers executed at all. gc might not even be run if the program execution does not warrant it. Thus any memory allocated during program execution might remain allocated after program termination, unless reclaimed by the OS or by other means.
48. There are also no guarantees on the order in which the objects will be garbage collected or on the order in which the finalizers are called. Therefore, the program should not make any decisions based on these assumptions.
49. An object is only eligible for garbage collection, if the only references to the object are from other objects that are also eligible for garbage collection. That is, an object can become eligible for garbage collection even if there are references pointing to the object, as long as the objects with the references are also eligible for garbage collection.
50. Circular references do not prevent objects from being garbage collected.
51. We can set the reference variables to null, hinting the gc to garbage collect the objects referred by the variables. Even if we do that, the object may not be gc-ed if it's attached to a listener. (Typical in case of AWT components) Remember to remove the listener first.
52. All objects have a finalize method. It is inherited from the Object class.
53. finalize method is used to release system resources other than memory. (such as file handles and network connections) The order in which finalize methods are called may not reflect the order in which objects are created. Don't rely on it. This is the signature of the finalize method.

    protected void finalize() throws Throwable { }

    In the descendents this method can be protected or public. Descendents can restrict the exception list that can be thrown by this method.
54. finalize is called only once for an object. If any exception is thrown in finalize, the object is still eligible for garbage collection (at the discretion of gc)
55. gc keeps track of unreachable objects and garbage-collects them, but an unreachable object can become reachable again by letting know other objects of its existence from its finalize method (when called by gc). This 'resurrection' can be done only once, since finalize is called only one for an object.
56. finalize can be called explicitly, but it does not garbage collect the object.
57. finalize can be overloaded, but only the method with original finalize signature will be called by gc.
58. finalize is not implicitly chained. A finalize method in sub-class should call finalize in super class explicitly as its last action for proper functioning. But compiler doesn't enforce this check.
59. System.runFinalization can be used to run the finalizers (which have not been executed before) for the objects eligible for garbage collection.
60. The following table specifies the color coding of javadoc standard. (May be not applicable to 1.2)

Member	Color
Instance method	Red
Static method	Green
Final variable	Blue
Constructor	Yellow

Chapter 2 Operators and assignments

1. Unary operators.
   - 1.1 Increment and Decrement operators ++ --
     We have postfix and prefix notation. In post-fix notation value of the variable/expression is <u>modified after</u> the value is taken for the execution of statement. In prefix notation, value of the variable/expression is <u>modified before</u> the value is taken for the execution of statement.

     x = 5; y = 0; y = x++; Result will be x = 6, y = 5
     x = 5; y = 0; y = ++x; Result will be x = 6, y = 6

     Implicit narrowing conversion is done, when applied to byte, short or char.

   - 1.2 Unary minus and unary plus + -
     + has no effect than to stress positivity.
     - negates an expression's value. (2's complement for integral expressions)
   - 1.3 Negation !
     Inverts the value of a boolean expression.
   - 1.4 Complement ~
     Inverts the bit pattern of an integral expression. (1's complement – 0s to 1s and 1s to 0s)
     Cannot be applied to non-integral types.
   - 1.5 Cast ()
     Persuades compiler to allow certain assignments. Extensive checking is done at compile and runtime to ensure type-safety.
2. Arithmetic operators - *, /, %, +, -
   - Can be applied to all numeric types.
   - Can be applied to only the numeric types, except '+' – it can be applied to Strings as well.
   - All arithmetic operations are done at least with 'int'. (If types are smaller, promotion happens. Result will be of a type at least as wide as the wide type of operands)
   - Accuracy is lost silently when arithmetic overflow/error occurs. Result is a nonsense value.
   - Integer division by zero throws an exception.
   - Floating point arithmetic always loses precision. The following code fragment returns false.
     float f = 1.0 F / 3.0 F; if (A * 3.0 F == 1.0 F) return true; else return false;
   - % - reduce the magnitude of LHS by the magnitude of RHS. (continuous subtraction)
   - % - sign of the result entirely determined by sign of LHS
   - 5 % 0 throws an ArithmeticException.
   - Floating point calculations can produce NaN (square root of a negative no) or Infinity (division by zero). Float and Double wrapper classes have named constants for NaN and infinities.
   - NaN's are non-ordinal for comparisons. x == Float.NaN won't work. Use Float.IsNaN(x) But equals method on wrapper objects(Double or Float) with NaN values compares Nan's correctly.
   - Infinities are ordinal. X == Double.POSITIVE_INFINITY will give expected result.
   - + also performs String concatenation (when any operand in an expression is a String). The language itself overloads this operator. toString method of non-String object operands are called to perform concatenation. In case of primitives, a wrapper object is created with the primitive value and toString method of that object is called. ("Vel" + 3 will work.)
   - Be aware of associativity when multiple operands are involved.
     System.out.println(1 + 2 + "3"); // Prints 33
     System.out.println("1" + 2 + 3); // Prints 123
3. Shift operators - <<, >>, >>>
   - << performs a signed left shift. 0 bits are brought in from the right. Sign bit (MSB) is preserved. Value becomes old value * 2 ^ x where x is no of bits shifted.
   - >> performs a signed right shift. Sign bit is brought in from the left. (0 if positive, 1 if negative. Value becomes old value / 2 ^ x where x is no of bits shifted. Also called arithmetic right shift.
   - >>> performs an unsigned logical right shift. 0 bits are brought in from the left. This operator exists since Java doesn't provide an unsigned data type (except char). >>> changes the sign of a

negative number to be positive. So don't use it with negative numbers, if you want to preserve the sign. Also don't use it with types smaller than int. (Since types smaller than int are promoted to an int before any shift operation and the result is cast down again, so the end result is unpredictable.)

- Shift operators can be applied to only integral types.
- -1 >> 1 is –1, not 0. This differs from simple division by 2. We can think of it as shift operation rounding down.
- 1 << 31 will become the minimum value that an int can represent. (Value becomes negative, after this operation, if you do a signed right shift sign bit is brought in from the left and the value remains negative.)
- Negative numbers are represented in two's complement notation. (Take one's complement and add 1 to get two's complement)
- Shift operators never shift more than the number of bits the type of result can have. (i.e. int 32, long 64) RHS operand is reduced to RHS % x where x is no of bits in type of result.

            int x;
            x = x >> 33; // Here actually what happens is x >> 1

4. Comparison operators – all return boolean type.
    4.1 Ordinal comparisons - <, <=, > , >=
        - Only operate on numeric types. Test the relative value of the numeric operands.
        - Arithmetic promotions apply. char can be compared to float.
    4.2 Object type comparison – instanceof
        - Tests the class of an object at runtime. Checking is done at compile and runtime same as the cast operator.
        - Returns true if the object denoted by LHS reference can be cast to RHS type.
        - LHS should be an object reference expression, variable or an array reference.
        - RHS should be a class (abstract classes are fine), an interface or an array type, castable to LHS object reference. Compiler error if LHS & RHS are unrelated.
        - Can't use java.lang.Class or its String name as RHS.
        - Returns true if LHS is a class or subclass of RHS class
        - Returns true if LHS implements RHS interface.
        - Returns true if LHS is an array reference and of type RHS.
        - x instanceof Component[] – legal.
        - x instanceof [] – illegal. Can't test for 'any array of any type'
        - Returns false if LHS is null, no exceptions are thrown.
        - If x instanceof Y is not allowed by compiler, then Y y = (Y) x is not a valid cast expression. If x instanceof Y is allowed and returns false, the above cast is valid but throws a ClassCastException at runtime. If x instanceof Y returns true, the above cast is valid and runs fine.
    4.3 Equality comparisons - ==, !=
        - For primitives it's a straightforward value comparison. (promotions apply)
        - For object references, this doesn't make much sense. Use equals method for meaningful comparisons. (Make sure that the class implements equals in a meaningful way, like for X.equals(Y) to be true, Y instance of X must be true as well)
        - For String literals, == will return true, this is because of compiler optimization.

5. Bit-wise operators - &, ^, |
    - Operate on numeric and boolean operands.
    - & - AND operator, both bits must be 1 to produce 1.
    - | - OR operator, any one bit can be 1 to produce 1.
    - ^ - XOR operator, any one bit can be 1, but not both, to produce 1.
    - In case of booleans true is 1, false is 0.
    - Can't cast any other type to boolean.

6. Short-circuit logical operators - &&, ||
    - Operate only on boolean types.
    - RHS might not be evaluated (hence the name short-circuit), if the result can be determined only by looking at LHS.

- false && X is always false.
- true || X is always true.
- RHS is evaluated only if the result is not certain from the LHS.
- That's why there's no logical XOR operator. Both bits need to be known to calculate the result.
- Short-circuiting doesn't change the result of the operation. But side effects might be changed. (i.e. some statements in RHS might not be executed, if short-circuit happens. Be careful)

7. Ternary operator
   - Format a = x ? b : c ;
   - x should be a boolean expression.
   - Based on x, either b or c is evaluated. Both are never evaluated.
   - b will be assigned to a if x is true, else c is assigned to a.
   - b and c should be assignment compatible to a.
   - b and c are made identical during the operation according to promotions.

8. Assignment operators.
   - Simple assignment =.
   - op= calculate and assign operators(extended assignment operators)
   - *=, /=, %=, +=, -=
   - x += y means x = x + y. But x is evaluated only once. Be aware.
   - Assignment of reference variables copies the reference value, not the object body.
   - Assignment has value, value of LHS after assignment. So a = b = c = 0 is legal. c = 0 is executed first, and the value of the assignment (0) assigned to b, then the value of that assignment (again 0) is assigned to a.
   - Extended assignment operators do an implicit cast. (Useful when applied to byte, short or char)
     ```
     byte b = 10;
     b = b + 10; // Won't compile, explicit cast reqd since the expression evaluates to an int
     b += 10; // OK, += does an implicit cast from int to byte
     ```

9. General
   - In Java, No overflow or underflow of integers happens. i.e. The values wrap around. Adding 1 to the maximum int value results in the minimum value.
   - Always keep in mind that operands are evaluated from left to right, and the operations are executed in the order of precedence and associativity.
   - Unary Postfix operators and all binary operators (except assignment operators) have left to right assoiciativity.
   - All unary operators (except postfix operators), assignment operators, ternary operator, object creation and cast operators have right to left assoiciativity.
   - Inspect the following code.
     ```
     public class Precedence {
       final public static void main(String args[]) {
             int i = 0;
             i = i++;
             i = i++;
             i = i++;
             System.out.println(i); // prints 0, since = operator has the lowest precedence.

             int array[] = new int[5];
             int index = 0;
             array[index] = index = 3; // 1st element get assigned to 3, not the 4th element

             for (int c = 0; c < array.length; c++)
               System.out.println(array[c]);
             System.out.println("index is " + index); // prints 3
       }
     }
     ```

Type of Operators	Operators	Associativity
Postfix operators	[] . (parameters) ++ --	Left to Right
Prefix Unary operators	++ -- + - ~ !	Right to Left
Object creation and cast	new (type)	Right to Left
Multiplication/Division/Modulus	* / %	Left to Right
Addition/Subtraction	+ -	Left to Right
Shift	>> >>> <<	Left to Right
Relational	< <= > >= instanceof	Left to Right
Equality	== !=	Left to Right
Bit-wise/Boolean AND	&	Left to Right
Bit-wise/Boolean XOR	^	Left to Right
Bit-wise/Boolean OR	|	Left to Right
Logical AND (Short-circuit or Conditional)	&&	Left to Right
Logical OR (Short-circuit or Conditional)	||	Left to Right
Ternary	? :	Right to Left
Assignment	= += -= *= /= %= <<= >>= >>>= &= ^= |=	Right to Left

Chapter 3 Modifiers

1. Modifiers are Java keywords that provide information to compiler about the nature of the code, data and classes.
2. Access modifiers – public, protected, private
   - Only applied to class level variables. Method variables are visible only inside the method.
   - Can be applied to class itself (only to inner classes declared at class level, no such thing as protected or private top level class)
   - Can be applied to methods and constructors.
   - If a class is accessible, it doesn't mean, the members are also accessible. Members' accessibility determines what is accessible and what is not. But if the class is not accessible, the members are not accessible, even though they are declared public.
   - If no access modifier is specified, then the accessibility is default package visibility. All classes in the same package can access the feature. It's called as friendly access. But friendly is not a Java keyword. Same directory is same package in Java's consideration.
   - 'private' means only the class can access it, not even sub-classes. So, it'll cause access denial to a sub-class's own variable/method.
   - These modifiers dictate, which <u>classes</u> can access the features. An <u>instance</u> of a <u>class</u> can access the private features of another <u>instance</u> of the same <u>class</u>.
   - 'protected' means all classes in the same package (like default) and sub-classes in any package can access the features. But a subclass in another package can access the protected members in the super-class via only the references of subclass or its subclasses. A subclass in the same package doesn't have this restriction. This ensures that classes from other packages are accessing only the members that are part of their inheritance hierarchy.
   - Methods cannot be overridden to be more private. Only the direction shown in following figure is permitted from parent classes to sub-classes.

     private → friendly (default) → protected → public

     Parent classes   Sub-classes

3. final
   - final features cannot be changed.
   - final classes cannot be sub-classed.
   - final variables cannot be changed. (Either a value has to be specified at declaration or an assignment statement can appear only once).
   - final methods cannot be overridden.
   - Method arguments marked final are read-only. Compiler error, if trying to assign values to final arguments inside the method.
   - Member variables marked final are not initialized by default. They have to be explicitly assigned a value at declaration or in an initializer block. Static finals must be assigned to a value in a static initializer block, instance finals must be assigned a value in an instance initializer or in every constructor. Otherwise the compiler will complain.
   - Final variables that are not assigned a value at the declaration and method arguments that are marked final are called blank final variables. They can be assigned a value at most once.
   - Local variables can be declared final as well.
4. abstract
   - Can be applied to classes and methods.
   - For deferring implementation to sub-classes.
   - Opposite of final, final can't be sub-classed, abstract must be sub-classed.
   - A class should be declared abstract,
     1. if it has any abstract methods.
     2. if it doesn't provide implementation to any of the abstract methods it inherited
     3. if it doesn't provide implementation to any of the methods in an interface that it says implementing.

- Just terminate the abstract method signature with a ';', curly braces will give a compiler error.
- A class can be abstract even if it doesn't have any abstract methods.

5. static
   - Can be applied to nested classes, methods, variables, free floating code-block (static initializer)
   - Static variables are initialized at class load time. A class has only one copy of these variables.
   - Static methods can access only static variables. (They have no this)
   - Access by class name is a recommended way to access static methods/variables.
   - Static initializer code is run at class load time.
   - Static methods may not be overridden to be non-static.
   - Non-static methods may not be overridden to be static.
   - Abstract methods may not be static.
   - Local variables cannot be declared as static.
   - Actually, static methods are not participating in the usual overriding mechanism of invoking the methods based on the class of the object at runtime. Static method binding is done at compile time, so the method to be invoked is determined by the type of reference variable rather than the actual type of the object it holds at runtime.

     Let's say a sub-class has a static method which 'overrides' a static method in a parent class. If you have a reference variable of parent class type and you assign a child class object to that variable and invoke the static method, the method invoked will be the parent class method, not the child class method. The following code explains this.

```
public class StaticOverridingTest {
  public static void main(String s[]) {
        Child c = new Child();
        c.doStuff(); // This will invoke Child.doStuff()

        Parent p = new Parent();
        p.doStuff(); // This will invoke Parent.doStuff()

        p = c;
        p.doStuff(); // This will invoke Parent.doStuff(), rather than Child.doStuff()
  }
}

class Parent {
  static int x = 100;
  public static void doStuff() {
        System.out.println("In Parent..doStuff");
        System.out.println(x);
  }
}

class Child extends Parent {
  static int x = 200;
  public static void doStuff() {
        System.out.println("In Child..doStuff");
        System.out.println(x);
  }
}
```

6. native
   - Can be applied to methods only. (static methods also)
   - Written in a non-Java language, compiled for a single machine target type.
   - Java classes use lot of native methods for performance and for accessing hardware Java is not aware of.

- Native method signature should be terminated by a ';', curly braces will provide a compiler error.
- native doesn't affect access qualifiers. Native methods can be private.
- abstract can appear with native declaration. This forces the entire class to be abstract.(obviously)
- Can pass/return Java objects from native methods.
- System.loadLibrary is used in static initializer code to load native libraries. If the library is not loaded when the static method is called, an UnsatisfiedLinkError is thrown.

7. transient
   - Can be applied to class level variables only.(Local variables cannot be declared transient)
   - Transient variables may not be final or static.(But compiler allows the declaration, since it doesn't do any harm. Variables marked transient are never serialized. Static variables are not serialized anyway.)
   - Not stored as part of object's persistent state, i.e. not written out during serialization.
   - Can be used for security.

8. synchronized
   - Can be applied to methods or parts of methods only.
   - Used to control access to critical code in multi-threaded programs.

9. volatile
   - Can be applied to variables only.
   - Can be applied to static variables.
   - Cannot be applied to final variables.
   - Declaring a variable volatile indicates that it might be modified asynchronously, so that all threads will get the correct value of the variable.
   - Used in multi-processor environments.

Modifier	Class	Inner classes (Except local and anonymous classes)	Variable	Method	Constructor	Free floating Code block
public	Y	Y	Y	Y	Y	N
protected	N	Y	Y	Y	Y	N
(friendly) No access modifier	Y	Y (OK for all)	Y	Y	Y	N
private	N	Y	Y	Y	Y	N
final	Y	Y (Except anonymous classes)	Y	Y	N	N
abstract	Y	Y (Except anonymous classes)	N	Y	N	N
static	N	Y	Y	Y	N	Y (static initializer)
native	N	N	N	Y	N	N
transient	N	N	Y	N	N	N
synchronized	N	N	N	Y	N	Y (part of method, also need to specify an object on which a lock should be obtained)
volatile	N	N	Y	N	N	N

Chapter 4 Converting and Casting

Unary Numeric Promotion
Contexts:

- Operand of the unary arithmetic operators + and –
- Operand of the unary integer bit-wise complement operator ~
- During array creation, for example new int[x], where the dimension expression x must evaluate to an int value.
- Indexing array elements, for example table['a'], where the index expression must evaluate to an int value.
- Individual operands of the shift operators.

Binary numeric promotion
Contexts:

- Operands of arithmetic operators *, / , %, + and –
- Operands of relational operators <, <= , > and >=
- Numeric Operands of equality operators == and !=
- Integer Operands of bit-wise operators &, ^ and |

Conversion of Primitives
1. 3 types of conversion – assignment conversion, method call conversion and arithmetic promotion
2. boolean may not be converted to/from any non-boolean type.
3. Widening conversions accepted. Narrowing conversions rejected.
4. byte, short can't be converted to char and vice versa.
5. Arithmetic promotion
   5.1 Unary operators
   - if the operand is byte, short or char {
         convert it to int;
     }
      else {
         do nothing; no conversion needed;
     }
   5.2 Binary operators
   - if one operand is double {
         all double; convert the other operand to double;
     }
     else if one operand is float {
         all float; convert the other operand to float;
     }
     else if one operand is long {
         all long; convert the other operand to long;
     }
     else {
         all int; convert all to int;
     }
6. When assigning a literal value to a variable, the range of the variable's data type is checked against the value of the literal and assignment is allowed or compiler will produce an error.
       char c = 3; // this will compile, even though a numeric literal is by default an int since the range of char will accept the value
       int a = 3;
       char d = a; // this won't compile, since we're assigning an int to char
       char e = -1; // this also won't compile, since the value is not in the range of char
       float f = 1.3; // this won't compile, even though the value is within float range. Here range is not important, but precision is. 1.3 is by default a double, so a specific cast or f = 1.3f will work.
       float f = 1/3; // this will compile, since RHS evaluates to an int.
       Float f = 1.0 / 3.0; // this won't compile, since RHS evaluates to a double.

7. Also when assigning a final variable to a variable, even if the final variable's data type is wider than the variable, if the value is within the range of the variable an implicit conversion is done.

```
byte b;
final int a = 10;
b = a; // Legal, since value of 'a' is determinable and within range of b
final int x = a;
b = x; // Legal, since value of 'x' is determinable and within range of b
int y;
final int z = y;
b = z; // Illegal, since value of 'z' is not determinable
```

8. Method call conversions always look for the exact data type or a wider one in the method signatures. They will not do narrowing conversions to resolve methods, instead we will get a compile error.

Here is the figure of allowable primitive conversion.

$$byte \rightarrow short \rightarrow int \rightarrow long \rightarrow float \rightarrow double$$
$$\uparrow$$
$$char$$
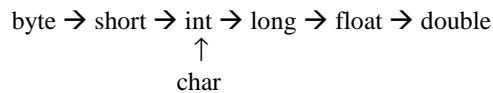
Casting of Primitives
9. Needed with narrowing conversions. Use with care – radical information loss. Also can be used with widening conversions, to improve the clarity of the code.
10. Can cast any non-boolean type to another non-boolean type.
11. Cannot cast a boolean or to a boolean type.

Conversion of Object references
12. Three types of reference variables to denote objects - class, interface or array type.
13. Two kinds of objects can be created – class or array.
14. Two types of conversion – assignment and method call.
15. Permitted if the direction of the conversion is 'up' the inheritance hierarchy. Means that types can be assigned/substituted to only super-types – super-classes or interfaces. Not the other way around, explicit casting is needed for that.
16. Interfaces can be used as types when declaring variables, so they participate in the object reference conversion. But we cannot instantiate an interface, since it is abstract and doesn't provide any implementation. These variables can be used to hold objects of classes that implement the interface. The reason for having interfaces as types may be, I think, several unrelated classes may implement the same interface and if there's a need to deal with them collectively one way of treating them may be an array of the interface type that they implement.
17. Primitive arrays can be converted to only the arrays of the same primitive type. They cannot be converted to another type of primitive array. Only object reference arrays can be converted / cast.
18. Primitive arrays can be converted to an Object reference, but not to an Object[] reference. This is because all arrays (primitive arrays and Object[]) are extended from Object.

Casting of Object references
19. Allows super-types to be assigned to subtypes. Extensive checks done both at compile and runtime. At compile time, class of the object may not be known, so at runtime if checks fail, a ClassCastException is thrown.
20. Cast operator, instanceof operator and the == operator behave the same way in allowing references to be the operands of them. You cannot cast or apply instanceof or compare *unrelated references*, *sibling references or any incompatible references.*

Compile-time Rules
- When old and new types are classes, one class must be the sub-class of the other.

- When old and new types are arrays, both must contain reference types and it must be legal to cast between those types (primitive arrays cannot be cast, conversion possible only between same type of primitive arrays).
- We can always cast between an interface and a non-final object.

Run-time rules
- If new type is a class, the class of the expression being converted must be new type or extend new type.
- If new type is an interface, the class of the expression being converted must implement the interface.

An Object reference can be converted to: (java.lang.Object)
- an Object reference
- a Cloneable interface reference, with casting, with runtime check
- any class reference, with casting, with runtime check
- any array referenece, with casting, with runtime check
- any interface reference, with casting, with runtime check

A Class type reference can be converted to:
- any super-class type reference, (including Object)
- any sub-class type reference, with casting, with runtime check
- an interface reference, if the class implements that interface
- any interface reference, with casting, with runtime check (except if the class is final and doesn't implement the interface)

An Interface reference can be converted to:
- an Object reference
- a super-interface reference
- any interface/class reference with casting, with runtime check (except if the class is final and doesn't implement the interface)

A Primitive Array reference can be converted to:
- an Object reference
- a Cloneable interface reference
- a primitive array reference of the same type

An Object Array reference can be converted to:
- an Object reference
- a Cloneable interface reference
- a super-class Array reference, including an Object Array reference
- any sub-class Array reference with casting, with runtime check

Chapter 5 Flow Control and Exceptions

- Unreachable statements produce a compile-time error.
    - while (false) { x = 3; } // won't compile
    - for (;false;) { x =3; } // won't compile
    - if (false) {x = 3; } // will compile, to provide the ability to conditionally compile the code.
- Local variables already declared in an enclosing block, therefore visible in a nested block cannot be re-declared inside the nested block.
- A local variable in a block may be re-declared in another local block, if the blocks are disjoint.
- Method parameters cannot be re-declared.

1. Loop constructs
   - 3 constructs – for, while, do
   - All loops are controlled by a boolean expression.
   - In while and for, the test occurs at the top, so if the test fails at the first time, body of the loop might not be executed at all.
   - In do, test occurs at the bottom, so the body is executed <u>at least once</u>.
   - In for, we can declare multiple variables in the first part of the loop separated by commas, also we can have multiple statements in the third part separated by commas.
   - In the first section of for statement, we can have a list of declaration statements <u>or</u> a list of expression statements, but not both. We cannot mix them.
   - All expressions in the third section of for statement will always execute, even if the first expression makes the loop condition false. There is no short –circuit here.
2. Selection Statements
   - if takes a boolean arguments. Parenthesis required. else part is optional. else if structure provides multiple selective branching.
   - switch takes an argument of byte, short, char or int.(assignment compatible to int)
   - case value should be a constant expression that can be evaluated at compile time.
   - Compiler checks each case value against the range of the switch expression's data type. The following code won't compile.
           byte b;
           switch (b) {
             case 200: // 200 not in range of byte
             default:
           }
   - We need to place a break statement in each case block to prevent the execution to fall through other case blocks. But this is not a part of switch statement and not enforced by the compiler.
   - We can have multiple case statements execute the same code. Just list them one by one.
   - default case can be placed anywhere. It'll be executed *only if* none of the case values match.
   - switch can be nested. Nested case labels are independent, don't clash with outer case labels.
   - Empty switch construct is a valid construct. But any statement within the switch block should come under a case label or the default case label.
3. Branching statements
   - break statement can be used with any kind of loop or a switch statement or just a labeled block.
   - continue statement can be used with only a loop (any kind of loop).
   - Loops can have labels. We can use break and continue statements to branch out of multiple levels of nested loops using labels.
   - Names of the labels follow the same rules as the name of the variables.(Identifiers)
   - Labels can have the same name, as long as they don't enclose one another.
   - There is no restriction against using the same identifier as a label and as the name of a package, class, interface, method, field, parameter, or local variable.
4. Exception Handling
   - An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

- There are 3 main advantages for exceptions:
  1. Separates error handling code from "regular" code
  2. Propagating errors up the call stack (without tedious programming)
  3. Grouping error types and error differentiation
- An exception causes a jump to the end of try block. If the exception occurred in a method called from a try block, the called method is abandoned.
- If there's a catch block for the occurred exception or a parent class of the exception, the exception is now considered handled.
- At least one 'catch' block or one 'finally' block must accompany a 'try' statement. If all 3 blocks are present, the order is important. (try/catch/finally)
- finally and catch can come only with try, they cannot appear on their own.
- Regardless of whether or not an exception occurred or whether or not it was handled, if there is a finally block, it'll be executed always. (Even if there is a return statement in try block).
- System.exit() and error conditions are the only exceptions where finally block is not executed.
- If there was no exception or the exception was handled, execution continues at the statement after the try/catch/finally blocks.
- If the exception is not handled, the process repeats looking for next enclosing try block up the call hierarchy. If this search reaches the top level of the hierarchy (the point at which the thread was created), then the thread is killed and message stack trace is dumped to System.err.
- Use throw new xxxException() to throw an exception. If the thrown object is null, a NullPointerException will be thrown at the handler.
- If an exception handler re-throws an exception (throw in a catch block), same rules apply. Either you need to have a try/catch within the catch or specify the entire method as throwing the exception that's being re-thrown in the catch block. Catch blocks at the same level will not handle the exceptions thrown in a catch block – it needs its own handlers.
- The method fillInStackTrace() in Throwable class throws a Throwable object. It will be useful when re-throwing an exception or error.
- The Java language requires that methods either *catch* or *specify* all checked exceptions that can be thrown within the scope of that method.
- All objects of type java.lang.Exception are checked exceptions. (Except the classes under java.lang.RuntimeException) If any method that contains lines of code that might throw checked exceptions, compiler checks whether you've handled the exceptions or you've declared the methods as throwing the exceptions. Hence the name checked exceptions.
- If there's no code in try block that may throw exceptions specified in the catch blocks, compiler will produce an error. (This is not the case for super-class Exception)
- Java.lang.RuntimeException and java.lang.Error need not be handled or declared.
- An overriding method may not throw a checked exception unless the overridden method also throws that exception or a super-class of that exception. In other words, an overriding method may not throw checked exceptions that are not thrown by the overridden method. If we allow the overriding methods in sub-classes to throw more general exceptions than the overridden method in the parent class, then the compiler has no way of checking the exceptions the sub-class might throw. (If we declared a parent class variable and at runtime it refers to sub-class object) This violates the concept of checked exceptions and the sub-classes would be able to by-pass the enforced checks done by the compiler for checked exceptions. This should not be allowed.

Here is the exception hierarchy.

```
Object
  |
  |
Throwable
  |        |
  |        |
  |        |
  |      Error
  |
  |
Exception
```

Exception-->ClassNotFoundException, ClassNotSupportedException, IllegalAccessException, InstantiationException, IterruptedException, NoSuchMethodException, <u>RuntimeException</u>, AWTException, <u>IOException</u>

RuntimeException-->EmptyStackException, NoSuchElementException, ArithmeticException, ArrayStoreException, ClassCastException, <u>IllegalArgumentException</u>, IllegalMonitorStateException, <u>IndexOutOfBoundsException</u>, NegativeArraySizeException, NullPointerException, SecurityException.

IllegalArgumentException-->IllegalThreadStateException, NumberFormatException

IndexOutOfBoundsException-->ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException

IOException-->EOFException, FileNotFoundException, InterruptedIOException, UTFDataFormatException, MalformedURLException, ProtocolException, SockException, UnknownHostException, UnknownServiceException.

Implementing OO relationships
- "is a" relationship is implemented by inheritance (extends keyword)
- "has a" relationship is implemented by providing the class with member variables.

Overloading and Overriding
- Overloading is an example of polymorphism. (operational / parametric)
- Overriding is an example of runtime polymorphism (inclusive)
- A method can have the same name as another method in the same class, provided it forms either a valid overload or override

Overloading	Overriding
Signature has to be different. Just a difference in return type is not enough.	Signature has to be the same. (including the return type)
Accessibility may vary freely.	Overriding methods cannot be more private than the overridden methods.
Exception list may vary freely.	Overriding methods may not throw more checked exceptions than the overridden methods.
Just the name is reused. Methods are independent methods. Resolved at compile-time based on method signature.	Related directly to sub-classing. Overrides the parent class method. Resolved at run-time based on type of the object.
Can call each other by providing appropriate argument list.	Overriding method can call overridden method by super.methodName(), this can be used only to access the immediate super-class's method. super.super won't work. Also, a class outside the inheritance hierarchy can't use this technique.
Methods can be static or non-static. Since the methods are independent, it doesn't matter. But if two methods have the same signature, declaring one as static and another as non-static does not provide a valid overload. It's a compile time error.	static methods don't participate in overriding, since they are resolved at compile time based on the type of reference variable. A static method in a sub-class can't use 'super' (for the same reason that it can't use 'this' for) Remember that a static method can't be overridden to be non-static and a non-static method can't be overridden to be static. In other words, a static method and a non-static method cannot have the same name and signature (if signatures are different, it would have formed a valid overload)
There's no limit on number of overloaded methods a class can have.	Each parent class method may be overridden at most once in any sub-class. (That is, you cannot have two identical methods in the same class)

- Variables can also be overridden, it's known as shadowing or hiding. But, member variable references are resolved at compile-time. So at the runtime, if the class of the object referred by a parent class reference variable, is in fact a sub-class having a shadowing member variable, only the parent class variable is accessed, since it's already resolved at compile time based on the reference variable type. Only methods are resolved at run-time.

```
public class Shadow {
  public static void main(String s[]) {
        S1 s1 = new S1();
        S2 s2 = new S2();

        System.out.println(s1.s); // prints S1
        System.out.println(s1.getS()); // prints S1
```

```
            System.out.println(s2.s); // prints S2
            System.out.println(s2.getS()); // prints S2

            s1 = s2;

            System.out.println(s1.s); // prints S1, not S2 -
                                      // since variable is resolved at compile time

            System.out.println(s1.getS()); // prints S2 -
                                           // since method is resolved at run time
    }
}

class S1 {
  public String s = "S1";

  public String getS() {
        return s;
  }
}

class S2 extends S1{
  public String s = "S2";

  public String getS() {
        return s;
  }
}
```

In the above code, if we didn't have the overriding getS() method in the sub-class and if we call the method from sub-class reference variable, the method will return only the super-class member variable value. For explanation, see the following point.

- Also, methods access variables only in context of the class of the object they belong to. If a sub-class method calls explicitly a super class method, the super class method always will access the super-class variable. Super class methods will not access the shadowing variables declared in subclasses because they don't know about them. (When an object is created, instances of all its super-classes are also created.) But the method accessed will be again subject to dynamic lookup. It is always decided at runtime which implementation is called. (Only static methods are resolved at compile-time)

```
public class Shadow2 {
  String s = "main";
  public static void main(String s[]) {
        S2 s2 = new S2();
        s2.display();  // Produces an output – S1, S2

        S1 s1 = new S1();
        System.out.println(s1.getS()); // prints S1
        System.out.println(s2.getS()); // prints S1 – since super-class method always
                                       // accesses super-class variable

  }
}

class S1 {
```

```
    String s = "S1";
    public String getS() {
            return s;
    }
    void display() {
            System.out.println(s);
    }
}

class S2 extends S1{
    String s = "S2";
    void display() {
            super.display();   // Prints S1
            System.out.println(s); // prints S2
    }
}
```

- With OO languages, the class of the object may not be known at compile-time (by virtue of inheritance). JVM from the start is designed to support OO. So, the JVM insures that the method called will be from the real class of the object (not with the variable type declared). This is accomplished by virtual method invocation (late binding). Compiler will form the argument list and produce one method invocation instruction – its job is over. The job of identifying and calling the proper target code is performed by JVM.
- JVM knows about the variable's real type at any time since when it allocates memory for an object, it also marks the type with it. Objects always know 'who they are'. This is the basis of instanceof operator.
- Sub-classes can use super keyword to access the shadowed variables in super-classes. This technique allows for accessing only the immediate super-class. super.super is not valid. But casting the 'this' reference to classes up above the hierarchy will do the trick. By this way, variables in super-classes above any level can be accessed from a sub-class, since variables are resolved at compile time, when we cast the 'this' reference to a super-super-class, the compiler binds the super-super-class variable. But this technique is not possible with methods since methods are resolved **always** at runtime, and the method gets called depends on the type of object, not the type of reference variable. So **it is not at all possible** to access a method in a super-super-class from a subclass.

```
public class ShadowTest {
        public static void main(String s[]){
                new STChild().demo();
        }
}
class STGrandParent {
        double wealth = 50000.00;
        public double getWealth() {
                System.out.println("GrandParent-" + wealth);
                return wealth;
        }
}
class STParent extends STGrandParent {
        double wealth = 100000.00;
        public double getWealth() {
                System.out.println("Parent-" + wealth);
                return wealth;
        }
}
class STChild extends STParent {
        double wealth = 200000.00;

        public double getWealth() {
                System.out.println("Child-" + wealth);
                return wealth;
```

```
        }
        public void demo() {
                getWealth(); // Calls Child method
                super.getWealth(); // Calls Parent method
                //super.super.getWealth(); // Compiler error, GrandParent method cannot be accessed
                ((STParent)this).getWealth(); // Calls Child method, due to dynamic method lookup
                ((STGrandParent)this).getWealth(); // Calls Child method, due to dynamic method
                                        // lookup

                System.out.println(wealth); // Prints Child wealth
                System.out.println(super.wealth); // Prints Parent wealth
                System.out.println(((STParent)(this)).wealth); // Prints Parent wealth
                System.out.println(((STGrandParent)(this)).wealth); // Prints GrandParent wealth
        }
}
```

- An inherited method, which was not abstract on the super-class, can be declared abstract in a sub-class (thereby making the sub-class abstract). There is no restriction. In the same token, a subclass can be declared abstract regardless of whether the super-class was abstract or not.
- Private members are not inherited, but they do exist in the sub-classes. Since the private methods are not inherited, they cannot be overridden. A method in a subclass with the same signature as a private method in the super-class is essentially a new method, independent from super-class, since the private method in the super-class is not visible in the sub-class.

```
public class PrivateTest {
        public static void main(String s[]){
                new PTSuper().hi();   // Prints always Super
                new PTSub().hi();      // Prints Super when subclass doesn't have hi method
                                        // Prints Sub when subclass has hi method
                PTSuper sup;
                sup = new PTSub();
                sup.hi();    // Prints Super when subclass doesn't have hi method
                                // Prints Sub when subclass has hi method
        }
}

class PTSuper {
        public void hi() { // Super-class implementation always calls superclass hello
                hello();
        }
        private void hello() {  // This method is not inherited by subclasses, but exists in them.
                                // Commenting out both the methods in the subclass show this.
                                // The test will then print "hello-Super" for all three calls
                                // i.e. Always the super-class implementations are called
                System.out.println("hello-Super");
        }
}

class PTSub extends PTSuper {
        public void hi() { // This method overrides super-class hi, calls subclass hello
                try {
                        hello();
                }
                catch(Exception e) { }

        }

        void hello() throws Exception {   // This method is independent from super-class hello
                                        // Evident from, it's allowed to throw Exception
                System.out.println("hello-Sub");
        }
}
```

- Private methods are not overridden, so calls to private methods are resolved at compile time and not subject to dynamic method lookup. See the following example.

```
public class Poly {
        public static void main(String args[]) {
                PolyA ref1 = new PolyC();
                PolyB ref2 = (PolyB)ref1;

                System.out.println(ref2.g());        // This prints 1
                                                     // If f() is not private in PolyB, then prints 2
        }
}

class PolyA {
        private int f() { return 0; }
        public int g() { return 3; }
}

class PolyB extends PolyA {
        private int f() { return 1; }
        public int g() { return f(); }
}

class PolyC extends PolyB {
        public int f() { return 2; }
}
```

Constructors and Sub-classing
- Constructors are not inherited as normal methods, they have to be defined in the class itself.
- If you define no constructors at all, then the compiler provides a default constructor with no arguments. Even if, you define one constructor, this default is not provided.
- We can't compile a sub-class if the immediate super-class doesn't have a no argument default constructor, and sub-class constructors are not calling super or this explicitly (and expect the compiler to insert an implicit super() call)
- A constructor can call other overloaded constructors by 'this (arguments)'. If you use this, it must be the first statement in the constructor. This construct can be used only from within a constructor.
- A constructor can't call the same constructor from within. Compiler will say ' recursive constructor invocation'
- A constructor can call the parent class constructor explicitly by using 'super (arguments)'. If you do this, it must be first the statement in the constructor. This construct can be used only from within a constructor.
- Obviously, we can't use both this and super in the same constructor. If compiler sees a this or super, it won't insert a default call to super().
- Constructors can't have a return type. A method with a class name, but with a return type is not considered a constructor, but just a method by compiler. Expect trick questions using this.
- Constructor body can have an empty return statement. Though void cannot be specified with the constructor signature, empty return statement is acceptable.
- Only modifiers that a constructor can have are the accessibility modifiers.
- Constructors cannot be overridden, since they are not inherited.
- Initializers are used in initialization of objects and classes and to define constants in interfaces. These initializers are :
  1. Static and Instance variable initializer expressions.
     Literals and method calls to initialize variables. Static variables can be initialized only by static method calls.
     Cannot pass on the checked exceptions. Must catch and handle them.
  2. Static initializer blocks.
     Used to initialize static variables and load native libraries.
     Cannot pass on the checked exceptions. Must catch and handle them.
  3. Instance initializer blocks.
     Used to factor out code that is common to all the constructors.
     Also useful with anonymous classes since they cannot have constructors.
     All constructors must declare the uncaught checked exceptions, if any.

Instance Initializers in anonymous classes can throw any exception.

- In all the initializers, forward referencing of variables is not allowed. Forward referencing of methods is allowed.
- Order of code execution (when creating an object) is a bit tricky.
  1. static variables initialization.
  2. static initializer block execution. (in the order of declaration, if multiple blocks found)
  3. constructor header (super or this – implicit or explicit)
  4. instance variables initialization / instance initializer block(s) execution
  5. rest of the code in the constructor

Interfaces

- All methods in an interface are implicitly public, abstract, and never static.
- All variables in an interface are implicitly static, public, final. They cannot be transient or volatile. A class can shadow the variables it inherits from an interface, with its own variables.
- A top-level interface itself cannot be declared as static or final since it doesn't make sense.
- Declaring parameters to be final is at method's discretion, this is not part of method signature.
- Same case with final, synchronized, native. Classes can declare the methods to be final, synchronized or native whereas in an interface they cannot be specified like that. (These are implementation details, interface need not worry about this)
- But classes cannot implement an interface method with a static method.
- If an interface specifies an exception list for a method, then the class implementing the interface need not declare the method with the exception list. (Overriding methods can specify sub-set of overridden method's exceptions, here none is a sub-set). But if the interface didn't specify any exception list for a method, then the class cannot throw any exceptions.
- All interface methods should have public accessibility when implemented in class.
- Interfaces cannot be declared final, since they are implicitly abstract.
- A class can implement two interfaces that have a method with the same signature or variables with the same name.

Inner Classes

- A class can be declared in any scope. Classes defined inside of other classes are known as **nested classes**. There are four categories of nested classes.
  1. Top-level nested classes / interfaces
     - Declared as a class member with static modifier.
     - Just like other static features of a class. Can be accessed / instantiated without an instance of the outer class. Can access only static members of outer class. Can't access instance variables or methods.
     - Very much like any-other package level class / interface. Provide an extension to packaging by the modified naming scheme at the top level.
     - Classes can declare both static and non-static members.
     - Any accessibility modifier can be specified.
     - Interfaces are implicitly static (static modifier also can be specified). They can have any accessibility modifier. There are no non-static inner, local or anonymous interfaces.
  2. Non-static inner classes
     - Declared as a class member without static.
     - An instance of a non-static inner class can exist only with an instance of its enclosing class. So it always has to be created within a context of an outer instance.
     - Just like other non-static features of a class. Can access all the features (even private) of the enclosing outer class. Have an implicit reference to the enclosing instance.
     - Cannot have any static members.
     - Can have any access modifier.
  3. Local classes
     - Defined inside a block (could be a method, a constructor, a local block, a static initializer or an instance initializer). Cannot be specified with static modifier.
     - Cannot have any access modifier (since they are effectively local to the block)

- Cannot declare any static members.(Even declared in a static context)
- Can access all the features of the enclosing class (because they are defined inside the method of the class) but can access only final variables defined inside the method (including method arguments). This is because the class can outlive the method, but the method local variables will go out of scope – in case of final variables, compiler makes a copy of those variables to be used by the class. (New meaning for final)
- Since the names of local classes are not visible outside the local context, references of these classes cannot be declared outside. So their functionality could be accessed only via super-class references (either interfaces or classes). Objects of those class types are created inside methods and returned as super-class type references to the outside world. This is the reason that they can only access final variables within the local block. That way, the value of the variable can be always made available to the objects returned from the local context to outside world.
- Cannot be specified with static modifier. But if they are declared inside a static context such as a static method or a static initializer, they become static classes. They can only access static members of the enclosing class and local final variables. But this doesn't mean they cannot access any non-static features inherited from super classes. These features are their own, obtained via the inheritance hierarchy. They can be accessed normally with 'this' or 'super'.

4. Anonymous classes
   - Anonymous classes are defined where they are constructed. They can be created wherever a reference expression can be used.
   - Anonymous classes cannot have explicit constructors. Instance initializers can be used to achieve the functionality of a constructor.
   - Typically used for creating objects on the fly.
   - Anonymous classes can implement an interface (implicit extension of Object) or explicitly extend a class. Cannot do both.
     Syntax: new interface name() { } or new class name() { }
   - Keywords implements and extends are not used in anonymous classes.
   - Abstract classes can be specified in the creation of an anonymous class. The new class is a concrete class, which automatically extends the abstract class.
   - Discussion for local classes on static/non-static context, accessing enclosing variables, and declaring static variables also holds good for anonymous classes. In other words, anonymous classes cannot be specified with static, but based on the context, they could become static classes. In any case, anonymous classes are not allowed to declare static members. Based on the context, non-static/static features of outer classes are available to anonymous classes. Local final variables are always available to them.

- One enclosing class can have multiple instances of inner classes.
- Inner classes can have synchronous methods. But calling those methods obtains the lock for inner object only not the outer object. If you need to synchronize an inner class method based on outer object, outer object lock must be obtained explicitly. Locks on inner object and outer object are independent.
- Nested classes can extend any class or can implement any interface. No restrictions.
- All nested classes (except anonymous classes) can be abstract or final.
- Classes can be nested to any depth. Top-level static classes can be nested only within other static top-level classes or interfaces. Deeply nested classes also have access to all variables of the outer-most enclosing class (as well the immediate enclosing class's)
- Member inner classes can be forward referenced. Local inner classes cannot be.
- An inner class variable can shadow an outer class variable. In this case, an outer class variable can be referred as (outerclassname.this.variablename).
- Outer class variables are accessible within the inner class, but they are not inherited. They don't become members of the inner class. This is different from inheritance. (Outer class cannot be referred using 'super', and outer class variables cannot be accessed using 'this')
- An inner class variable can shadow an outer class variable. If the inner class is sub-classed within the same outer class, the variable has to be qualified explicitly in the sub-class. To

fully qualify the variable, use classname.this.variablename. If we don't correctly qualify the variable, a compiler error will occur. (Note that this does not happen in multiple levels of inheritance where an upper-most super-class's variable is silently shadowed by the most recent super-class variable or in multiple levels of nested inner classes where an inner-most class's variable silently shadows an outer-most class's variable. Problem comes only when these two hierarchy chains (inheritance and containment) clash.)

- If the inner class is sub-classed outside of the outer class (only possible with top-level nested classes) explicit qualification is not needed (it becomes regular class inheritance)

```java
// Example 1

public class InnerInnerTest {
  public static void main(String s[]) {
      new Outer().new Inner().new InnerInner().new InnerInnerInner().doSomething();
      new Outer().new InnerChild().doSomething();
      new Outer2().new Inner2().new InnerInner2().doSomething();
      new InnerChild2().doSomething();

  }
}

class Outer {
  String name = "Vel";

  class Inner {
      String name = "Sharmi";

      class InnerInner {

        class InnerInnerInner {

                public void doSomething() {

                  // No problem in accessing without full qualification,
                  // inner-most class variable shadows the outer-most class variable
                  System.out.println(name); // Prints "Sharmi"
                  System.out.println(Outer.this.name); // Prints "Vel", explicit reference to Outer

// error, variable is not inherited from the outer class, it can be just accessible
//                System.out.println(this.name);
//                System.out.println(InnerInner.this.name);
//                System.out.println(InnerInnerInner.this.name);

// error, super cannot be used to access outer class.
// super will always refer the parent, in this case Object
//                System.out.println(super.name);

                  System.out.println(Inner.this.name); // Prints "Sharmi", Inner has declared 'name'
                }
           }
         }
   }

   /* This is an inner class extending an inner class in the same scope */
   class InnerChild extends Inner {
       public void doSomething() {
// compiler error, explicit qualifier needed
// 'name' is inherited from Inner, Outer's 'name' is also in scope
//       System.out.println(name);
         System.out.println(Outer.this.name); // prints "Vel", explicit reference to Outer
         System.out.println(super.name); // prints "Sharmi", Inner has declared 'name'
         System.out.println(this.name); // prints "Sharmi", name is inherited by InnerChild
       }
   }
}

class Outer2 {
```

```java
    static String name = "Vel";
    static class Inner2 {
        static String name = "Sharmi";

        class InnerInner2 {
          public void doSomething() {
            System.out.println(name); // prints "Sharmi", inner-most hides outer-most
            System.out.println(Outer2.name); // prints "Vel", explicit reference to Outer2's static variable
//          System.out.println(this.name); // error, 'name' is not inherited
//          System.out.println(super.name); // error, super refers to Object
          }
        }
    }

}

/* This is a stand-alone class extending an inner class */
class InnerChild2 extends Outer2.Inner2 {
    public void doSomething() {
      System.out.println(name); // prints "Sharmi", Inner2's name is inherited
      System.out.println(Outer2.name); // prints "Vel", explicit reference to Outer2's static variable
      System.out.println(super.name); // prints "Sharmi", Inner2 has declared 'name'
      System.out.println(this.name); // prints "Sharmi", name is inherited by InnerChild2
    }
}
```

// Example 2

```java
public class InnerTest2 {
  public static void main(String s[]) {

        new OuterClass().doSomething(10, 20);

// This is legal
//      OuterClass.InnerClass ic = new OuterClass().new InnerClass();
//      ic.doSomething();

// Compiler error, local inner classes cannot be accessed from outside
//      OuterClass.LocalInnerClass lic = new OuterClass().new LocalInnerClass();
//      lic.doSomething();

        new OuterClass().doAnonymous();

  }
}

class OuterClass {
  final int a = 100;
  private String secret = "Nothing serious";

  public void doSomething(int arg, final int fa) {
        final int x = 100;
        int y = 200;

        System.out.println(this.getClass() + " - in doSomething");
        System.out.print("a = " + a + " secret = " + secret + " arg = " + arg + " fa = " + fa);
        System.out.println(" x = " + x + " y = " + y);

// Compiler error, forward reference of local inner class
//      new LocalInnerClass().doSomething();

        abstract class AncestorLocalInnerClass { } // inner class can be abstract

        final class LocalInnerClass extends AncestorLocalInnerClass { // can be final
          public void doSomething() {
            System.out.println(this.getClass() + " - in doSomething");
            System.out.print("a = " + a );
            System.out.print(" secret = " + secret);
//          System.out.print(" arg = " + arg);  // Compiler error, accessing non-final argument
```

28

```java
                System.out.print(" fa = " + fa);
                System.out.println(" x = " + x);
//               System.out.println(" y = " + y); // Compiler error, accessing non-final variable
              }
            }

            new InnerClass().doSomething(); // forward reference fine for member inner class
            new LocalInnerClass().doSomething();
  }

  abstract class AncestorInnerClass { }

  interface InnerInterface { final int someConstant = 999;} // inner interface

  class InnerClass extends AncestorInnerClass implements InnerInterface {
    public void doSomething() {
      System.out.println(this.getClass() + " - in doSomething");
      System.out.println("a = " + a + " secret = " + secret + " someConstant = " + someConstant);
    }
  }

  public void doAnonymous() {
            // Anonymous class implementing the inner interface
            System.out.println((new InnerInterface() { }).someConstant);

            // Anonymous class extending the inner class
            ( new InnerClass() {
                    public void doSomething() {
                      secret = "secret is changed";
                      super.doSomething();
                    }
            } ).doSomething();
  }
}
```

29

Entity	Declaration Context	Accessibility Modifiers	Outer instance	Direct Access to enclosing context	Defines static or non-static members
Package level class	As package member	Public or default	No	N/A	Both static and non-static
Top level nested class (static)	As static class member	All	No	Static members in enclosing context	Both static and non-static
Non static inner class	As non-static class member	All	Yes	All members in enclosing context	Only non-static
Local class (non-static)	In block with non-static context	None	Yes	All members in enclosing context + local final variables	Only non-static
Local class (static)	In block with static context	None	No	Static members in enclosing context + local final variables	Only non-static
Anonymous class (non-static)	In block with non-static context	None	Yes	All members in enclosing context + local final variables	Only non-static
Anonymous class (static)	In block with static context	None	No	Static members in enclosing context + local final variables	Only non-static
Package level interface	As package member	Public or default	No	N/A	Static variables and non-static method prototypes
Top level nested interface (static)	As static class member	All	No	Static members in enclosing context	Static variables and non-static method prototypes

Chapter 7 Threads

- Java is fundamentally multi-threaded.
- Every thread corresponds to an instance of java.lang.Thread class or a sub-class.
- A thread becomes eligible to run, when its start() method is called. Thread scheduler co-ordinates between the threads and allows them to run.
- When a thread begins execution, the scheduler calls its run method.
    Signature of run method – public void run()
- When a thread returns from its run method (or stop method is called – deprecated in 1.2), its dead. It cannot be restarted, but its methods can be called. (it's just an object no more in a running state)
- If start is called again on a dead thread, IllegalThreadStateException is thrown.
- When a thread is in running state, it may move out of that state for various reasons. When it becomes eligible for execution again, thread scheduler allows it to run.
- There are two ways to implement threads.
1. Extend Thread class
    - Create a new class, extending the Thread class.
    - Provide a public void run method, otherwise empty run in Thread class will be executed.
    - Create an instance of the new class.
    - Call start method on the instance (don't call run – it will be executed on the same thread)

2. Implement Runnable interface
    - Create a new class implementing the Runnable interface.
    - Provide a public void run method.
    - Create an instance of this class.
    - Create a Thread, passing the instance as a target – new Thread(object)
    - Target should implement Runnable, Thread class implements it, so it can be a target itself.
    - Call the start method on the Thread.

- JVM creates one user thread for running a program. This thread is called main thread. The main method of the class is called from the main thread. It dies when the main method ends. If other user threads have been spawned from the main thread, program keeps running even if main thread dies. Basically a program runs until all the user threads (non-daemon threads) are dead.
- A thread can be designated as a daemon thread by calling setDaemon(boolean) method. This method should be called before the thread is started, otherwise IllegalThreadStateException will be thrown.
- A thread spawned by a daemon thread is a daemon thread.
- Threads have priorities. Thread class have constants MAX_PRIORITY (10), MIN_PRIORITY (1), NORM_PRIORITY (5)
- A newly created thread gets its priority from the creating thread. Normally it'll be NORM_PRIORITY.
- getPriority and setPriority are the methods to deal with priority of threads.
- Java leaves the implementation of thread scheduling to JVM developers. Two types of scheduling can be done.
1. Pre-emptive Scheduling.
    Ways for a thread to leave running state -
    - It can cease to be ready to execute (by calling a blocking i/o method)
    - It can get pre-empted by a high-priority thread, which becomes ready to execute.
    - It can explicitly call a thread-scheduling method such as wait or suspend.

    - Solaris JVM's are pre-emptive.
    - Windows JVM's were pre-emptive until Java 1.0.2

2. Time-sliced or Round Robin Scheduling
    - A thread is only allowed to execute for a certain amount of time. After that, it has to contend for the CPU (virtual CPU, JVM) time with other threads.
    - This prevents a high-priority thread mono-policing the CPU.

- The drawback with this scheduling is – it creates a non-deterministic system – at any point in time, you cannot tell which thread is running and how long it may continue to run.

  - Mactinosh JVM's
  - Windows JVM's after Java 1.0.2
- Different states of a thread:
1. Yielding
   - Yield is a static method. Operates on current thread.
   - Moves the thread from running to ready state.
   - If there are no threads in ready state, the yielded thread may continue execution, otherwise it may have to compete with the other threads to run.
   - Run the threads that are doing time-consuming operations with a low priority and call yield periodically from those threads to avoid those threads locking up the CPU.
2. Sleeping
   - Sleep is also a static method.
   - Sleeps for a certain amount of time. (passing time without doing anything and w/o using CPU)
   - Two overloaded versions – one with milliseconds, one with milliseconds and nanoseconds.
   - Throws an InterruptedException.(must be caught)
   - After the time expires, the sleeping thread goes to ready state. It may not execute immediately after the time expires. If there are other threads in ready state, it may have to compete with those threads to run. The correct statement is the sleeping thread would execute *some time after* the specified time period has elapsed.
   - If interrupt method is invoked on a sleeping thread, the thread moves to ready state. The next time it begins running, it executes the InterruptedException handler.
3. Suspending
   - Suspend and resume are instance methods and are deprecated in 1.2
   - A thread that receives a suspend call, goes to suspended state and stays there until it receives a resume call on it.
   - A thread can suspend it itself, or another thread can suspend it.
   - But, a thread can be resumed only by another thread.
   - Calling resume on a thread that is not suspended has no effect.
   - Compiler won't warn you if suspend and resume are successive statements, although the thread may not be able to be restarted.
4. Blocking
   - Methods that are performing I/O have to wait for some occurrence in the outside world to happen before they can proceed. This behavior is blocking.
   - If a method needs to wait an indeterminable amount of time until some I/O takes place, then the thread should graciously step out of the CPU. All Java I/O methods behave this way.
   - A thread can also become blocked, if it failed to acquire the lock of a monitor.
5. Waiting
   - wait, notify and notifyAll methods are not called on Thread, they're called on Object. Because the object is the one which controls the threads in this case. It asks the threads to wait and then notifies when its state changes. It's called a monitor.
   - Wait puts an executing thread into waiting state.(to the monitor's waiting pool)
   - Notify moves one thread in the monitor's waiting pool to ready state. We cannot control which thread is being notified. notifyAll is recommended.
   - NotifyAll moves all threads in the monitor's waiting pool to ready.
   - These methods can only be called from synchronized code, or an IllegalMonitorStateException will be thrown. In other words, only the threads that obtained the object's lock can call these methods.

Locks, Monitors and Synchronization
- Every object has a lock (for every synchronized code block). At any moment, this lock is controlled by at most one thread.

- A thread that wants to execute an object's synchronized code must acquire the lock of the object. If it cannot acquire the lock, the thread goes into blocked state and comes to ready only when the object's lock is available.
- When a thread, which owns a lock, finishes executing the synchronized code, it gives up the lock.
- Monitor (a.k.a Semaphore) is an object that can block and revive threads, an object that controls client threads. Asks the client threads to wait and notifies them when the time is right to continue, based on its state. In strict Java terminology, any object that has some synchronized code is a monitor.
- 2 ways to synchronize:
  1. Synchronize the entire method
     - Declare the method to be synchronized - very common practice.
     - Thread should obtain the object's lock.
  2. Synchronize part of the method
     - Have to pass an arbitrary object which lock is to be obtained to execute the synchronized code block (part of a method).
     - We can specify "this" in place object, to obtain very brief locking – not very common.

- wait – points to remember
  - calling thread gives up CPU
  - calling thread gives up the lock
  - calling thread goes to monitor's waiting pool
  - wait also has a version with timeout in milliseconds. Use this if you're not sure when the current thread will get notified, this avoids the thread being stuck in wait state forever.
- notify – points to remember
  - one thread gets moved out of monitor's waiting pool to ready state
  - notifyAll moves all the threads to ready state
  - Thread gets to execute <u>must re-acquire</u> the lock of the monitor before it can proceed.
- Note the differences between blocked and waiting.

Blocked	Waiting
Thread is waiting to get a lock on the monitor. (or waiting for a blocking i/o method)	Thread has been asked to wait. (by means of wait method)
Caused by the thread tried to execute some synchronized code. (or a blocking i/o method)	The thread already acquired the lock and executed some synchronized code before coming across a wait call.
Can move to ready only when the lock is available. (or the i/o operation is complete)	Can move to ready only when it gets notified (by means of notify or notifyAll)

- Points for complex models:
  1. Always check monitor's state in a while loop, rather than in an if statement.
  2. Always call notifyAll, instead of notify.
- Class locks control the static methods.
- wait and sleep must be enclosed in a try/catch for InterruptedException.
- A single thread can obtain multiple locks on multiple objects (or on the same object)
- A thread owning the lock of an object can call other synchronous methods on the same object. (this is another lock) Other threads can't do that. They should wait to get the lock.
- Non-synchronous methods can be called at any time by any thread.
- Synchronous methods are re-entrant. So they can be called recursively.
- Synchronized methods can be overridden to be non-synchronous. synchronized behavior affects only the original class.
- Locks on inner/outer objects are independent. Getting a lock on outer object doesn't mean getting the lock on an inner object as well, that lock should be obtained separately.
- wait and notify should be called from synchronized code. This ensures that while calling these methods the thread always has the lock on the object. If you have wait/notify in non-synchronized code

compiler won't catch this. At runtime, if the thread doesn't have the lock while calling these methods, an IllegalMonitorStateException is thrown.

- Deadlocks can occur easily. e.g, Thread A locked Object A and waiting to get a lock on Object B, but Thread B locked Object B and waiting to get a lock on Object A. They'll be in this state forever.
- It's the programmer's responsibility to avoid the deadlock. Always get the locks in the same order.
- While 'suspended', the thread keeps the locks it obtained – so suspend is deprecated in 1.2
- Use of stop is also deprecated, instead use a flag in run method. Compiler won't warn you, if you have statements after a call to stop, even though they are not reachable.

Chapter 8 java.lang package

- Object class is the ultimate ancestor of all classes. If there is no extends clause, compiler inserts 'extends object'. The following methods are defined in Object class. All methods are public, if not specified otherwise.

Method	Description
Boolean equals(Object o)	just does a == comparison, override in descendents to provide meaningful comparison
final native void wait() final native void notify() final native void notifyAll()	Thread control. Two other versions of wait() accept timeout parameters and may throw InterruptedException.
native int hashcode()	Returns a hash code value for the object. If two objects are equal according to the equals method, then calling the hashCode method on each of the two objects must produce the same integer result.

protected Object clone() throws CloneNotSupportedException
 	
CloneNotSupportedException is a checked Exception
	Creates a new object of the same class as this object. It then initializes each of the new object's fields by assigning it the same value as the corresponding field in this object. **No constructor is called.**
	The clone method of class Object will only clone an object whose class indicates that it is willing for its instances to be cloned. A class indicates that its instances can be cloned by declaring that it implements the Cloneable interface. Also the method has to be made public to be called from outside the class.
	Arrays have a public clone method.
	      int ia[ ][ ] = { { 1 , 2}, null };
	      int ja[ ][ ] = (int[ ] [ ])ia.clone();
	A clone of a multidimensional array is shallow, which is to say that it creates only a single new array. Subarrays are shared, so ia and ja are different but ia[0] and ja[0] are same.

final native Class getClass()	Returns the runtime class of an object.
String toString	Returns the string representation of the object. Method in Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. Override to provide useful information.
protected void finalize() throws Throwable	Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. Any exception thrown by the finalize method causes the finalization of this object to be halted, but is otherwise ignored. The finalize method in Object does nothing. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

- Math class is final, cannot be sub-classed.
- Math constructor is private, cannot be instantiated.
- All constants and methods are public and static, just access using class name.
- Two constants PI and E are specified.
- Methods that implement Trigonometry functions are native.
- All Math trig functions take angle input in radians.
  Angle degrees * PI / 180 = Angle radians
- Order of floating/double values:
  -Infinity --> Negative Numbers/Fractions --> -0.0 --> +0.0 --> Positive Numbers/Fractions --> Infinity
- abs – int, long, float, double versions available
- floor – greatest integer smaller than this number (look below towards the floor)
- ceil – smallest integer greater than this number (look above towards the ceiling)
- For floor and ceil functions, if the argument is NaN or infinity or positive zero or negative zero or already a value equal to a mathematical integer, the result is the same as the argument.
- For ceil, if the argument is less than zero but greater than –1.0, then the result is a negative zero
- random – returns a double between 0.0(including) and 1.0(excluding)

- round returns a long for double, returns an int for float. (closest int or long value to the argument) The result is rounded to an integer by adding ½ , taking the floor of the result, and casting the result to type int / long.

      (int)Math.floor(a + 0.5f)
      (long)Math.floor(a + 0.5d)

- double rint(double) returns closest double equivalent to a mathematical integer. If two values are equal, it returns the even integer value. rint(2.7) is 3, rint(2.5) is 2.
- Math.min(-0.0, +0.0) returns –0.0, Math.max(-0.0, +0.0) returns 0.0, -0.0 == +0.0 returns true.
- For a NaN or a negative argument, sqrt returns a NaN.

- Every primitive type has a wrapper class (some names are different – Integer, Boolean, Character)
- Wrapper class objects are immutable.
- All Wrapper classes are final.
- All wrapper classes, except Character, have a constructor accepting string. A Boolean object, created by passing a string, will have a value of false for any input other than "true" (case doesn't matter).
- Numeric wrapper constructors will throw a NumberFormatException, if the passed string is not a valid number. (empty strings and null strings also throw this exception)
- equals also tests the class of the object, so even if an Integer object and a Long object are having the same value, equals will return false.
- NaN's can be tested successfully with equals method.

      Float f1 = new Float(Float.NaN);
      Float f2 = new Float(Float.NaN);
      System.out.println(""+ (f1 == f2)+" "+f1.equals(f2)+ " "+(Float.NaN == Float.NaN));

  The above code will print false true false.
- Numeric wrappers have 6 methods to return the numeric value – intValue(), longValue(), etc.
- valueOf method parses an input string (optionally accepts a radix in case of int and long) and returns a new instance of wrapper class, on which it was invoked. It's a static method. For empty/invalid/null strings it throws a NumberFormatException. For null strings valueOf in Float and Double classes throw NullPointerException.
- parseInt and parseLong return primitive int and long values respectively, parsing a string (optionally a radix). Throw a NumberFormatException for invalid/empty/null strings.
- Numeric wrappers have overloaded toString methods, which accept corresponding primitive values (also a radix in case of int,long) and return a string.
- Void class represents void primitive type. It's not instantiable. Just a placeholder class.

- Strings are immutable.
- They can be created from a literal, a byte array, a char array or a string buffer.
- Literals are taken from pool (for optimization), so == will return true, if two strings are pointing to the same literal.
- Anonymous String objects (literals) may have been optimized even across classes.
- A String created by new operator is always a different new object, even if it's created from a literal.
- But a String can specify that its contents should be placed in a pool of unique strings for possible reuse, by calling intern() method. In programs that do a lot of String comparisons, ensuring that all Strings are in the pool, allows to use == comparison rather than the equals() method, which is slower.
- All string operations (concat, trim, replace, substring etc) construct and return new strings.
- toUpperCase and toLowerCase will return the same string if no case conversion was needed.
- equals takes an object (String class also has a version of equals that accepts a String), equalsIgnoreCase takes a string.
- Passing null to indexOf or lastIndexOf will throw NullPointerException, passing empty string returns 0, passing a string that's not in the target string returns –1.
- trim method removes all leading and trailing white-space from a String and returns a new String. White-space means, all characters with value less than or equal to the space character – '\u0020'.
- String class is final.

- + and += operators are overloaded for Strings.
- reverse, append, insert are not String methods.

- String Buffers are mutable strings.
- StringBuffer is a final class.
- They can be created empty, from a string or with a capacity. An empty StringBuffer is created with 16-character capacity. A StringBuffer created from a String has the capacity of the length of String + 16. StringBuffers created with the specified capacity has the exact capacity specified. Since they can grow dynamically in size without bounds, capacity doesn't have much effect.
- append, insert, setCharAt, reverse are used to manipulate the string buffer.
- setLength changes the length, if the current content is larger than specified length, it's truncated. If it is smaller than the specified length, nulls are padded. This method doesn't affect the capacity.
- equals on StringBuffer does a shallow comparison. (same like ==) Will return true only if the objects are same. Don't use it to test content equality
- trim is not a StringBuffer method.
- There is no relationship between String and StringBuffer. Both extend Object class.

- String context means, '+' operator appearing with one String operand. String concatenation cannot be applied to StringBuffers.
  - A new String buffer is created.
  - All operands are appended (by calling toString method, if needed)
  - Finally a string is returned by calling toString on the String Buffer.

- String concatenation process will add a string with the value of "null", if an object reference is null and that object is appearing in a concatenation expression by itself. But if we try to access its members or methods, a NullPointerException is thrown. The same is true for arrays, array name is replaced with null, but trying to index it when it's null throws a NullPointerException.

Chapter 9 java.util package

- A collection (a.k.a bag or multiset) allows a group of objects to be treated as a single unit. Arbitrary objects can be stored, retrieved and manipulated as elements of these collections.
- Collections Framework presents a set of standard utility classes to manage such collections.
  1. It contains 'core interfaces' which allow collections to be manipulated independent of their implementations. These interfaces define the common functionality exhibited by collections and facilitate data exchange between collections.
  2. A small set of implementations that are concrete implementations of the core interfaces, providing data structures that a program can use.
  3. An assortment of algorithms to perform various operations such as, sorting and searching.
- Collections framework is interface based, collections are implemented according to their interface type, rather than by implementation types. By using the interfaces whenever collections of objects need to be handled, interoperability and interchangeability are achieved.
- By convention each of the collection implementation classes provide a constructor to create a collection based on the elements in the Collection object passed as argument. By the same token, Map implementations provide a constructor that accepts a Map argument. This allows the implementation of a collection (Collection/Map) to be changed. But **Collections and Maps are not interchangeable**.
- Interfaces and their *implementations* in Java 1.2

```
Collection
   |
   |__ Set (no dupes, null allowed based on implementation) → HashSet
   |        |
   |        |__ SortedSet (Ordered Set) → TreeSet
   |
   |__ List (ordered collection, dupes OK) → Vector, ArrayList, LinkedList

Map (key-value pairs, null allowed based on implementation) → HashTable, HashMap
   |
   |__ SortedMap (Ordered Map) → TreeMap
```

Interface	Description
Collection	A basic interface that defines the operations that all the classes that maintain collections of objects typically implement.
Set	Extends Collection, sets that maintain unique elements. Set interface is defined in terms of the equals operation
SortedSet	Extends Set, maintain the elements in a sorted order
List	Extends Collection, maintain elements in a sequential order, duplicates allowed.
Map	A basic interface that defines operations that classes that represent mappings of keys to values typically implement
SortedMap	Extends Map for maps that maintain their mappings in key order.

- Classes that implement the interfaces use different storage mechanisms.
1. Arrays
   Indexed access is faster. Makes insertion, deletion and growing the store more difficult.
2. Linked List
   Supports insertion, deletion and growing the store. But indexed access is slower.
3. Tree
   Supports insertion, deletion and growing the store. Indexed access is slower. But searching is faster.
4. Hashing
   Supports insertion, deletion and growing the store. Indexed access is slower. But searching is faster. However, requires the use of unique keys for storing data elements.

Data Structures used to implement	Interfaces				
	Set	**SortedSet**	**List**	**Map**	**SortedMap**
Hash Table	HashSet (Nulls OK)			HashMap (Nulls OK) HashTable (No Nulls)	
Resizable Array			ArrayList (Nulls OK) Vector(Nulls OK)		
Balanced Tree		TreeSet			TreeMap
Linked List			LinkedList (Nulls OK)		

- Some of the operations in the collection interfaces are optional, meaning that the implementing class may choose not to provide a proper implementation of such an operation. In such a case, an UnsupportedOperationException is thrown when that operation is invoked.

Interface		Methods	Description
Collection	Basic Operations	int size(); boolean isEmpty(); boolean contains(Object element); *boolean add(Object element);* *boolean remove(Object element);*	Used to query a collection about its contents, and add/remove elements. The add() and remove() methods return true if the collection was modified as a result of the operation. The contains() method checks for membership.
	Bulk Operations	boolean containsAll(Collection c); *boolean addAll(Collection c);* *boolean removeAll(Collection c);* *boolean retainAll(Collection c);* *void clear();*	Perform on a collection as a single unit. Operations are equivalent of set logic on arbitrary collections (not just sets). The addAll(), removeAll(), clear() and retainAll() methods are destructive.
	Array Operations	Object[] toArray(); Object[] toArray(Object a[]);	These methods combined with Arrays.asList() method provide the bridge between arrays and collections.
	Iterators	Iterator iterator(); Iterator is an interface which has these methods. boolean hasNext(); Object next(); *void remove();*	Returns an iterator, to iterate the collection. The remove() method is the only recommended way to remove elements from a collection during the iteration.
Set		No new methods defined.	The add() method returns false, if the element is already in the Set. No exceptions are thrown.
List	Element Access by Index	Object get(int index); Object set(int index, Object element); void add(int index, Object element); Object remove(int index); boolean addAll(int index, Collection c);	First index is 0, last index is size() – 1. An illegal index throws IndexOutOfBoundsException.
	Element Search	int indexOf(Object o); int lastIndexOf(Object o);	If the element is not found, return –1.
	List Iterators	ListIterator listIterator(); ListIterator listIterator(int index); ListIterator extends Iterator. It allows iteration in both directions.	ListIterator's additional methods: boolean hasPrevious(); boolean previous(); int nextIndex(); int prviousIndex(); *void set(Object o);* *void add(Object o);*

	Open Range View	List subList(int fromIndex, int toIndex);	Returns a range of the list from fromIndex (inclusive) to toIndex (exclusive). Changes to view are reflected in the list and vice versa.

Map	Basic Operations	*Object put(Object key, Object value);*
		Object get(Object key);
		Object remove(Object key);
		boolean containsKey(Object key);
		boolean containsValue(Object value);
		int size();
		boolean isEmpty();
			The put method replaces the old value, if the map previously contained a mapping for that key.
			The get method returns null, if the specified key couldn't be found in the map.

	Bulk Operations	*void putAll(Map t);* *void clear();*	putAll() adds all the elements from the specified map. clear() removes all the elements from the map.

	Collection Views	Set keySet();
		Collection values();
		Set entrySet();
		
		Note that the values () method, returns a Collection, not a set. Reason is, multiple unique keys can map to the same value.
			Provide different views on a Map. Changes to views are reflected in the map and vice versa.
			
			Each <key,value> pair is represented by an Object implementing Map.Entry interface.
			    Object getKey();
			    Object getValue();
			    Object setValue(Object value);

SortedSet	Range View Operations	SortedSet headSet(Object toElement); SortedSet tailSet(Object fromElement); SortedSet subSet(Object fromElement, Object toElement);	fromElement is inclusive, toElement is exclusive. The views present the elements sorted in the same order as the underlying sorted set.
	Min-Max Points	Object first(); Object last();	Return the first (lowest) and last (highest) elements.
	Comparator Access	Comparator comparator();	Returns the compartor associated with this SortedSet, or null if it uses natural ordering.
SortedMap	Range View Operations	SortedMap headMap(Object toKey); SortedSet tailMap(Object fromKey); SortedSet subMap(Object fromKey, Object toKey);	SortedMap is sorted with keys. fromKey is inclusive, toKey is exclusive. The views present the elements sorted in the same order as the underlying sorted map.
	Min-Max Points	Object firstKey(); Object lastKey();	Return the first (lowest) and last (highest) keys.
	Comparator Access	Comparator comparator();	Returns the compartor associated with this SortedMap, or null if it uses natural ordering.

- Sorting in SortedSets and SortedMaps can be implemented in two ways.
  1. Objects can specify their natural order by implementing **Comparable** interface. Many if the standard classes in Java API, such as wrapper classes, String, Date and File implement this interface. This interface defines a single method:
     - int compareTo(Object o) – returns negative, zero, positive if the current object is less than, equal to or greater than the specified object.

     In this case a natural comparator queries objects implementing Comparable about their natural order. Objects implementing this interface can be used:
     - As elements in a sorted set.
     - As keys in sorted map.
     - In lists which can be sorted automatically by the Collections.sort() method.
  2. Objects can be sorted by specific comparators, which implement **Comparator** interface. This interface defines the following method:
     - int compare(Object o1, Object o2) – returns negative, zero, positive if the first object is less than, equal to or greater than the second object. It is recommended that its implementation doesn't contradict the semantics of the equals() method.

     Specific Comparators can be specified in the constructors of SortedSets and SortedMaps.

- All classes provide a constructor to create an empty collection (corresponding to the class). HashSet, HashMap, HashTable can also be specified with an initial capacity as well as a load factor (the ratio of number of elements stored to its current capacity). Most of the time, default values provide acceptable performance.
- A Vector, like an array, contains items that can be accessed using an integer index. However, the size of a Vector can grow and shrink as needed to accommodate adding and removing items after the Vector has been created.
- Vector (5,10) means initial capacity 5, additional allocation (capacity increment) by 10.
- Stack extends Vector and implements a LIFO stack. With the usual push() and pop() methods, there is a peek() method to look at the object at the top of the stack without removing it from the stack.
- Dictionary is an obsolete class. HashTable extends dictionary. Elements are stored as key-value pairs.
- Vector and HashTable are the only classes that are thread-safe.
- ArrayList (does what Vector does), HashMap(does what HashTable does), LinkedList and TreeMap are new classes in Java 1.2
- In Java 1.2, Iterator duplicates the functionality of Enumeration. New implementations should consider Iterator.

- Collections is a class, Collection is an interface.
- Collections class consists exclusively of static methods that operate on or return collections.
- Sorting and Searching algorithms in the Collections class.
  - static int binarySearch(List list, Object key)
  - static void fill(List list, Object o)
  - static void shuffle(List list, Object o)
  - static void sort(List list)
- Factory methods to provide thread-safety and data immutability. These methods return synchronized (thread-safe) / immutable collections from the specified collections.
  - List safeList = Collections.synchronizedList(new LinkedList());
  - SortedMap fixedMap = Collections.unmodifiableSortedMap(new SortedMap());
- Constants to denote immutable empty collections in the Collections class:
  - EMPTY_SET, EMPTY_LIST and EMPTY_MAP.
- Collections class also has the following methods:

Method	Description
public static Set singleton(Object o)	Returns an immutable set containing only the specified object
public static List singletonList(Object o)	Returns an immutable list containing only the specified object
public static Map singletonMap(Object key, Object value)	Returns an immutable map containing only the specified key, value pair.
public static List nCopies (int n, Object o)	Returns an immutable list consisting of n copies of the specified object. The newly allocated data object is tiny (it contains a single reference to the data object). This method is useful in combination with the List.addAll method to grow lists.

- The class Arrays, provides useful algorithms that operate on arrays. It also provides the static asList() method, which can be used to create List views of arrays. Changes to the List view affects the array and vice versa. The List size is the array size and cannot be modified. The asList() method in the Arrays class and the toArray() method in the Collection interface provide the bridge between arrays and collections.
  - Set mySet = new HashSet(Arrays.asList(myArray));
  - String[] strArray = (String[]) mySet.toArray();
- All concrete implementations of the interfaces in java.util package are inherited from abstract implementations of the interfaces. For example, HashSet extends AbstractSet, which extends AbstractCollection. LinkedList extends AbstractList, which extends AbstractCollection. These abstract

implementations already provide most of the heavy machinery by implementing relevant interfaces, so that customized implementations of collections can be easily implemented using them.

- **BitSet** class implements a vector of bits that grows as needed. Each component of the bit set has a boolean value. The bits of a BitSet are indexed by nonnegative integers. Individual indexed bits can be examined, set, or cleared. One BitSet may be used to modify the contents of another BitSet through logical AND, logical inclusive OR, and logical exclusive OR operations.

    By default, all bits in the set initially have the value false. A BitSet has a size of 64, when created without specifying any size.

- **ConcurrentModificationException** exception (extends **RuntimeException**) may be thrown by methods that have detected concurrent modification of a backing object when such modification is not permissible.

    For example, it is not permssible for one thread to modify a Collection while another thread is iterating over it. In general, the results of the iteration are undefined under these circumstances. Some Iterator implementations (including those of all the collection implementations provided by the JDK) may choose to throw this exception if this behavior is detected. Iterators that do this are known as *fail-fast* iterators, as they fail quickly and cleanly, rather that risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Chapter 10 Components

- Java's building blocks for creating GUIs.
- All non-menu related components inherit from java.awt.Component, that provides basic support for event handling, controlling component size, color, font and drawing of components and their contents.
- Component class implements ImageObserver, MenuContainer and Serializable interfaces. So all AWT components can be serialized and can host pop-up menus.
- Component methods:

Controls	Methods / Description
Size	Dimension getSize() void setSize(int width, int height) void setSize(Dimension d)
Location	Point getLocation() void setLocation(int x, int y) void setLocation(Point p)
Size and Location	Rectangle getBounds() void setBounds (int x, int y, int width, int height) void setBounds (Rectangle r)
Color	void setForeground(Color c) void setBackground(Color c)
Font	void setFont(Font f) void setFont(Font f)
Visibility and Enabling	void setEnabled(boolean b) void setVisible(boolean b)

- Container class extends Component. This class defines methods for nesting components in a container.
    - Component add(Component comp)
    - Component add(Component comp, int index)
    - void add(Component comp, Object constraints)
    - void add(Component comp, Object constraints, int index)

    - void remove(int index)
    - void remove(Component comp)
    - void removeAll()
- The following are the containers:

Container	Description
Panel	• Provides intermediate level of spatial organization and containment. • Not a top-level window • Does not have title, border or menubar. • Can be recursively nested. • Default layout is Flow layout.
Applet	• Specialized Panel, run inside other applications (typically browsers) • Changing the size of an applet is allowed or forbidden depending on the browser. • Default layout is Flow layout.

Window	• Top-level window without a title, border or menus.
	• Seldom used directly. Subclasses (Frame and Dialog) are used.
	• Defines these methods:
	  • void pack() – Initiates layout management, window size might be changed as a result
	  • void show() – Makes the window visible, also brings it to front
	  • void dispose() – When a window is no longer needed, call this to free resources.

Frame	• Top-level window (optionally user-resizable and movable) with a title-bar, an icon and menus. • Typically the starting point of a GUI application. • Default layout is Border layout.
Dialog	• Top-level window (optionally user-resizable and movable) with a title-bar. • Doesn't have icons or menus. • Can be made modal. • A parent frame needs to be specified to create a Dialog. • Default layout is Border layout.
ScrollPane	• Can contain a single component. If the component is larger than the scrollpane, it acquires vertical / horizontal scrollbars as specified in the constructor. • SCROLLBARS_AS_NEEDED – default, if nothing specified • SCROLLBARS_ALWAYS • SCROLLBARS_NEVER

- Top-level containers (Window, Frame and Dialog) cannot be nested. They can contain other containers and other components.
- GUI components:

Component	Description	Constructors	Events
Button	• A button with a textual label.	new Button("Apply")	Action event.
Canvas	• No default appearance. • Can be sub-classed to create custom drawing areas.		Mouse, MouseMotion, Key events.
Checkbox	• Toggling check box. • Default initial state is false. • getState(), setState(boolean state) - methods • Can be grouped with a CheckboxGroup to provide radio behavior. • Checkboxgroup is not a subclass of Component. • Checkboxgroup provides these methods: getSelectedCheckbox and setSelectedCheckbox(Checkbox new)	Checkbox(String label) Checkbox(String label, boolean initialstate) Checkbox(String label, CheckBoxGroup group)	Item event
Choice	• A pull-down list • Can be populated by repeatedly calling addItem(String item) method. • Only the current choice is visible.		Item event
FileDialog	• Subclass of Dialog • Open or Save file dialog, modal • Dialog automatically removed, after user selects the file or hits cancel. • getFile(), getDirectory() methods can be used to get information about the selected file.	FileDialog(Frame parent, String title, int mode) Mode can be FileDialog.LOAD or FileDialog.SAVE	
Label	• Displays a single line of read-only non-selectable text • Alignment can be Label.LEFT, Label.RIGHT or Label.CENTER	Label() Label(String label) Label(String label, int align)	None

List	• Scrollable vertical list of text items.
	• No of visible rows can be specified, if not specified layout manager determines this.
	• Acquires a vertical scrollbar if needed.
	• List class methods:
	  • addItem(String), addItem(String, int index)
	  • getItem(int index), getItemCount()
	  • getRows() – no of visible rows
	  • int getSelectedIndex()
	  • int[] getSelectedIndexes()
	  • String getSelectedItem()
	  • String[] getSelectedItems()
		List()
		List(int nVisibleRows)
		List(int nVisibleRows, boolean multiSelectOK)
			Item event – selecting or deselecting
			
			Action event – double clicking

Scrollbar	• With the last form of constructor, calculate the spread as maxvalue – minvalue. Then the slider width is slidersize / spread times of scrollbar width.	Scrollbar() – a vertical scrollbar. Scrollbar(int orientation) Scrollbar(int orientation, int initialvalue, int slidersize, int minvalue, int maxvalue) Orientation can be Scrollbar.HORIZONTAL Scrollbar.VERTICAL	Adjustment event

TextField	• Extends TextComponent
	• Single line of edit / display of text.
	• Scrolled using arrow keys.
	• Depending on the font, number of displayable characters can vary.
	• But, never changes size once created.
	• Methods from TextComponent:
	  • String getSelectedText()
	  • String getText()
	  • void setEditable(boolean editable)
	  • void setText(String text)
		TextField() – empty field
		TextField(int ncols) – size
		TextField(String text) – initial text
		TextField(String text, int ncols) – initial text and size
			Text event
			
			Action event – Enter key is pressed.

TextArea	• Extends TextComponent
	• Multiple lines of edit/display of text.
	• Scrolled using arrow keys.
	• Can use the TextComponent methods specified above.
	• Scroll parameter in last constructor form could be
	  TextArea.SCROLLBARS_BOTH,
	  TextArea.SCROLLBARS_NONE,
	  TextArea.SCROLLBARS_HORIZONTAL_ONLY
	  TextArea.SCROLLBARS_VERTICAL_ONLY
		TextArea() – empty area
		TextArea(int nrows, int ncols) – size
		TextArea(String text) – initial text
		TextArea(String text, int nrows, int ncols) – initial text and size
		TextArea(String text, int nrows, int ncols, int scroll)
			Text event

- Pull-down menus are accessed via a menu bar, which can appear only on Frames.
- All menu related components inherit from java.awt.MenuComponent
- Steps to create and use a menu
  - Create an instance of MenuBar class
  - Attach it to the frame – using setMenubar() method of Frame
  - Create an instance of Menu and populate it by adding MenuItems, CheckboxMenuItems, separators and Menus. Use addSeparator() method to add separators. Use add() method to add other items.

- Attach the Menu to the MenuBar. Use add() method of Menubar to add a menu to it. Use setHelpMenu to set a particular menu to appear always as right-most menu.
- Menu(String label) – creates a Menu instance. Label is what displayed on the Menubar. If this menu is used as a pull-down sub-menu, label is the menu item's label.
- MenuItems generate Action Events.
- CheckboxMenuItems generate Item Events.

Chapter 11 Layout Managers

- Precise layout functionality is often performed and a repetitive task. By the principles of OOP, it should be done by classes dedicated to it. These classes are layout managers.
- Platform independence requires that we delegate the positioning and painting to layout managers. Even then, Java does not guarantee a button will look the same in different platforms.(w/o using Swing)
- Components are added to a container using add method. A layout manager is associated with the container to handle the positioning and appearance of the components.
- add method is overloaded. Constraints are used differently by different layout managers. Index can be used to add the component at a particular place. Default is –1 (i.e. at the end)
    - Component add(Component comp)
    - Component add(Component comp, int index)
    - void add(Component comp, Object constraints)
    - void add(Component comp, Object constraints, int index)
- setLayout is used to associate a layout manager to a container. Panel class has a constructor that takes a layout manager. getLayout returns the associated layout manager.
- It is recommended that the layout manager be associated with the container before any component is added. If we associate the layout manager after the components are added and the container is already made visible, the components appear as if they have been added by the previous layout manager (if none was associated before, then the default). Only subsequent operations (such as resizing) on the container use the new layout manager. But if the container was not made visible before the new layout is added, the components are re-laid out by the new layout manager.
- Positioning can be done manually by passing null to setLayout.
- Flow Layout Manager
    - Honors components preferred size.(Doesn't constraint height or width)
    - Arranges components in horizontal rows, if there's not enough space, it creates another row.
    - If the container is not big enough to show all the components, Flow Layout Manager does not resize the component, it just displays whatever can be displayed in the space the container has.
    - Justification (LEFT, RIGHT or CENTER) can be specified in the constructor of layout manager.
    - Default for applets and panels.
- Grid Layout Manager
    - Never honors the components' preferred size
    - Arranges the components in no of rows/columns specified in the constructor. Divides the space the container has into equal size cells that form a matrix of rows/columns.
    - Each component will take up a cell in the order in which it is added (left to right, row by row)
    - Each component will be of the same size (as the cell)
    - If a component is added when the grid is full, a new column is created and the entire container is re-laid out.
- Border Layout Manager
    - Divides the container into 5 regions – NORTH, SOUTH, EAST, WEST and CENTER
    - When adding a component, specify which region to add. If nothing is specified, CENTER is assumed by default.
    - Regions can be specified by the constant strings defined in BorderLayout (all upper case) or using Strings (Title case, like North, South etc)
    - NORTH and SOUTH components – height honored, but made as wide as the container. Used for toolbars and status bars.
    - EAST and WEST components – width honored, but made as tall as the container (after the space taken by NORTH, SOUTH components). Used for scrollbars.
    - CENTER takes up the left over space. If there are no other components, it gets all the space.
    - If no component is added to CENTER, container's background color is painted in that space.
    - Each region can display only one component. If another component is added, it hides the earlier component.

- Card Layout Manager
    - Draws in time rather than space. Only one component displayed at a time.

- Like a tabbed panel without tabs. (Can be used for wizards interface, i.e. by clicking next, displays the next component)
- Components added are given a name and methods on the CardLayout manager can be invoked to show the component using this name. Also the manager contains methods to iterate through the components. For all methods, the parent container should be specified.
    - first(Container parent)
    - next(Container parent)
    - previous(Container parent)
    - last(Container parent)
    - show(Container parent, String name)
- Component shown occupies the entire container. If it is smaller it is resized to fit the entire size of the container. No visual clue is given about the container has other components.
- Gridbag Layout Manager
    - Like the GridLayout manger uses a rectangular grid.
    - Flexible. Components can occupy multiple cells. Also the width and height of the cells need not be uniform. i.e A component may span multiple rows and columns but the region it occupies is always rectangular. Components can have different sizes (which is not the case with Grid layout)
    - Requires lot of constraints to be set for each component that is added.
    - GridBagConstraints class is used to specify the constraints.
    - Same GridBagConstraints object can be re-used by all the components.

Specify	Name of the constraints	Description	Default
Location	int gridx int gridy	Column and row positions of the upper left corner of the component in the grid. Added relative to the previous component if specified GridBagConstraints.RELATIVE	GridBagConstraints.RELATIVE in both directions
Dimension	int gridwidth int gridheight	Number of cells occupied by the component horizontally and vertically in the grid. GridBagConstraints.REMAINDER - specify this for last component GridBagConstraints.RELATIVE - specify this for next-to-last component	One cell in both directions
Growth Factor	double weigthx double weigthy	How to use the extra space if available.	0 for both, meaning that the area allocated for the component will not grow beyond the preferred size.
Anchoring	int anchor	Where a component should be placed within its display area. Constants defined in GridBagConstraints: CENTER, NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST, NORTHWEST	GridBagConstraints.CENTER
Filling	int fill	How the component is to stretch and fill its display area. Constants defined in GridBagConstraints: NONE, BOTH, HORIZONTAL, VERTICAL	GridBagConstraints.NONE
Padding	int ipadx int ipady	Internal padding added to each side of the component. Dimension of the component will grow to (width + 2 * ipadx) and (height + 2 * ipady)	0 pixels in either direction
Insets	Insets insets	External padding (border) around the component.	(0,0,0,0) (top, left, bottom, right)

Layout Manager	Description	Constructors	Constants	Default For
FlowLayout	Lays out the components in row-major order. Rows growing from left to right, top to bottom.	FlowLayout() - center aligned, 5 pixels gap both horizontally and vertically FlowLayout(int alignment) FlowLayout(int alignment, int hgap, int vgap)	LEFT CENTER RIGHT	Panel and its subclasses (Applet)
GridLayout	Lays out the components in a specified rectangular grid, from left to right in each row and filling rows from top to bottom.	GridLayout() – equivalent to GridLayout(1,0) GridLayout(int rows, int columns)	N/A	None
BorderLayout	Up to 5 components can be placed in particular locations: north, south, east, west and center.	BorderLayout() BorderLayout(int hgap, int vgap)	NORTH SOUTH EAST WEST CENTER	Window and its subclasses (Dialog and Frame)
CardLayout	Components are handled as a stack of indexed cards. Shows only one at a time.	CardLayout() CardLayout(int hgap, int vgap)	N/A	None
GridbagLayout	Customizable and flexible layout manager that lays out the components in a rectangular grid.	GridbagLayout()	Defined in GridBag Constraints class. See the above table	None

Chapter 12 Events

- Java 1.0's original outward rippling event model had shortcomings.
  - An event could only be handled by the component that originated the event or one of its containers.
  - No way to disable processing of irrelevant events.
- Java 1.1 introduced new "event delegation model".
  - A component may be told which objects should be notified when the component generates a particular kind of event.
  - If a component is not interested in an event type, those events won't be propagated.
- Both models are supported in Java 2, but the old model will eventually disappear. Both models should not be mixed in a single program. If we do that, the program is most likely to fail.
- Event delegation model's main concepts: Event classes, Event listeners, Explicit event enabling and Event adapter classes.

Event Classes

- Events are Java objects. All the pertinent information is encapsulated in that object. The super class of all events is **java.util.EventObject.**
- This **java.util.EventObject** class defines a method that returns the object that generated the event:
  - Object getSource()
- All events related to AWT are in **java.awt.event** package. AWTEvent is the abstract super class of all AWT events. This class defines a method that returns the ID of the event. All events define constants to represent the type of event.
  - int getID() – returns an int in the form of an integer value that identifies the type of event.
- It is useful to divide the event classes into Semantic events and Low-level events.
- Semantic Events –
  These classes are used for high-level semantic events, to represent user interaction with GUI.
  ActionEvent, AdjustmentEvent, ItemEvent, TextEvent

Event Class	Source	Event Types
ActionEvent	Button – when clicked List – when doubleclicked MenuItem – when clicked TextField – when Enter key is pressed	ACTION_PERFORMED
AdjustmentEvent	Scrollbar – when adjustments are made	ADJUSTMENT_VALUE_CHANGED
ItemEvent	CheckBox – when selected or deselected CheckboxMenuItem – same as checkbox Choice – when an item is selected or deselected List - when an item is selected or deselected	ITEM_STATE_CHANGED
TextEvent	TextField TextArea	TEXT_VALUE_CHANGED

Methods defined in the events to get the information about them.

Event Class	Method	Description
ActionEvent	String getActionCommand	Returns the command name associated with this action
	int getModifiers	Returns the sum of modifier constants corresponding to the keyboard modifiers held down during this action. SHIFT_MASK, ALT_MASK, CTRL_MASK, META_MASK
AdjustmentEvent	int getvalue	Returns the current value designated by the adjustable component
ItemEvent	Object getItem	Returns the object that was selected or deselected Label of the checkbox
	int getStateChange SELECTED DESELECTED	Returned value indicates whether it was a selection or a de-selection that took place, given by the two constants in ItemEvent.

- Low-level Events –
  These classes are used to represent low-level input or window operations. Several low-level events can constitute a single semantic event.
  ComponentEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, PaintEvent, WindowEvent

Event Class	Source	Event Types
ComponentEvent	All components	COMPONENT_SHOWN, COMPONENT_HIDDEN, COMPONENT_MOVED, COMPONENT_RESIZED AWT handles this event automatically. Programs should not handle this event.
ContainerEvent	All containers	COMPONENT_ADDED, COMPONENT_REMOVED AWT handles this event automatically. Programs should not handle this event.
FocusEvent	All components	FOCUS_GAINED, FOCUS_LOST Receiving focus means the component will receive all the keystrokes.
InputEvent	All components	This is an abstract class. Parent of KeyEvent and MouseEvent. Constants for key and mouse masks are defined in this class.
KeyEvent	All components	KEYPRESSED, KEYRELEASED, KEYTYPED (when a character is typed)
MouseEvent	All components	MOUSE_PRESSED, MOUSE_RELEASED, MOUSE_CLICKED, MOUSE_DRAGGED, MOUSE_MOVED, MOUSE_ENTERED, MOUSE_EXITED
PaintEvent	All components	This event occurs when a component should have its paint()/update() methods invoked. AWT handles this event automatically. Programs should not handle this event. This event is not supposed to be handled by the event listener model. Components should override paint/update methods to get rendered.
WindowEvent	All windows	This event is generated when an important operation is performed on a window. WINDOW_OPENED, WINDOW_CLOSING, WINDOW_CLOSED, WINDOW_ICONIFIED, WINDOW_DEICONIFIED, WINDOW_ACTIVATED, WINDOW_DEACTIVATED

Methods defined in the events to get the information about them.

Event Class	Method	Description
ComponentEvent	Component getComponent	Returns a reference to the same object as getSource, but the returned reference is of type Component.
ContainerEvent	Container getContainer	Returns the container where this event originated.
	Component getChild	Returns the child component that was added or removed in this event
FocusEvent	boolean isTemporary	Determines whether the loss of focus is permanent or temporary
InputEvent	long getWhen	Returns the time the event has taken place.
	int getModifiers	Returns the modifiers flag for this event.
	void consume	Consumes this event so that it will not be processed in the default manner by the source that originated it.
KeyEvent	int getKeyCode	For KEY_PRESSED or KEY_RELEASED events, this method can be used to get the integer key-code associated with the key. Key-codes are defined as constants is KeyEvent class.
	char getKeyChar	For KEY_TYPED events, this method returns the Unicode character that was generated by the keystroke.
MouseEvent	int getX	Return the position of the mouse within the originated component at the time the event took place
	int getY	
	Point getPoint	
	int getClickCount	Returns the number of mouse clicks.
WindowEvent	Window getWindow	Returns a reference to the Window object that caused the event to be generated.

Event Listeners

- Each listener interface extends **java.util.EventListener** interface.
- There are 11 listener interfaces corresponding to particular events. Any class that wants to handle an event should implement the corresponding interface. Listener interface methods are passed the event object that has all the information about the event occurred.
- Then the listener classes should be registered with the component that is the source/originator of the event by calling the addXXXListener method on the component. Listeners are unregistered by calling removeXXXListener method on the component.
- A component may have multiple listeners for any event type.
- A component can be its own listener if it implements the necessary interface. Or it can handle its events by implementing the processEvent method. (This is discussed in explicit event enabling section)
- All registered listeners with the component are notified (by invoking the methods passing the event object). But the order of notification is not guaranteed (even if the same component is registered as its own listener). Also the notification is not guaranteed to occur on the same thread. Listeners should take cautions not to corrupt the shared data. Access to any data shared between the listeners should be synchronized.
- Same listener object can implement multiple listener interfaces.
- Event listeners are usually implemented as anonymous classes.

Event Type	Event Source	Listener Registration and removal methods provided by the source	Event Listener Interface implemented by a listener
ActionEvent	Button List MenuItem TextField	addActionListener removeActionListner	ActionListener
AdjustmentEvent	Scrollbar	addAdjustmentListener removeAdjustmentListner	AdjustmentListener
ItemEvent	Choice List Checkbox CheckboxMenuItem	addItemListener removeItemListner	ItemListener
TextEvent	TextField TextArea	addTextListener removeTextListner	TextListener
ComponentEvent	Component	add ComponentListener remove ComponentListner	ComponentListener
ContainerEvent	Container	addContainerListener removeContainerListner	ContainerListener
FocusEvent	Component	addFocusListener removeFocusListner	FocusListener
KeyEvent	Component	addKeyListener removeKeyListner	KeyListener
MouseEvent	Component	addMouseListener removeMouseListner	MouseListener
		addMouseMotionListener removeMouseMotionListner	MouseMotionListener
WindowEvent	Window	addWindowListener removeWindowListner	WindowListener

Event Listener interfaces and their methods:

Event Listener Interface	Event Listener Methods
ActionListener	void actionPerformed(ActionEvent evt)
AdjustmentListener	void adjustmentValueChanged(AdjustmentEvent evt)
ItemListener	void itemStateChanged(ItemEvent evt)
TextListener	void textValueChanged(TextEvent evt)
ComponentListener	void componentHidden(ComponentEvent evt) void componentShown(ComponentEvent evt) void componentMoved(ComponentEvent evt) void componentResized(ComponentEvent evt)
ContainerListener	void componentAdded(ContainerEvent evt) void componentRemoved(ContainerEvent evt)
FocusListener	void focusGained(FocusEvent evt) void focusLost(FocusEvent evt)
KeyListener	void keyPressed(KeyEvent evt) void keyReleased(KeyEvent evt) void keyTyped(KeyEvent evt)
MouseListener	void mouseClicked(MouseEvent evt) void mouseReleased(MouseEvent evt) void mousePressed(MouseEvent evt) void mouseEntered(MouseEvent evt) void mouseExited(MouseEvent evt)
MouseMotionListener	void mouseDragged(MouseEvent evt) void mouseMoved(MouseEvent evt)
WindowListener	void windowActivated(WindowEvent evt) void windowDeactivated(WindowEvent evt) void windowIconified(WindowEvent evt) void windowDeiconified(WindowEvent evt) void windowClosing(WindowEvent evt) void windowClosed(WindowEvent evt) void windowOpened(WindowEvent evt)

Event Adapters
- Event Adapters are convenient classes implementing the event listener interfaces. They provide empty bodies for the listener interface methods, so we can implement only the methods of interest without providing empty implementation. They are useful when implementing low-level event listeners.
- There are 7 event adapter classes, one each for one low-level event listener interface.
- Obviously, in semantic event listener interfaces, there is only one method, so there is no need for event adapters.
- Event adapters are usually implemented as anonymous classes.

Explicit Event Enabling
How events are produced and handled?
- OS dispatches events to JVM. How much low-level processing is done by OS or JVM depends on the type of the component. In case of Swing components JVM handles the low-level events.
- JVM creates event objects and passes them to the components.
- If the event is enabled for that component, processEvent method in that component (inherited from java.awt.Component) is called. Default behavior of this method is to delegate the processing to more

specific processXXXEvent method. Then this processXXXEvent method invokes appropriate methods in all registered listeners of this event.

- All the registered listeners of the event for the component are notified. But the order is not guaranteed.
- This delegation model works well for pre-defined components. If the component is customized by sub-classing another component, then it has the opportunity to handle its own events by implementing appropriate processXXXEvent methods or the processEvent method itself.
- To handle its own events, the subclass component must explicitly enable all events of interest. This is done by calling enableEvents method with appropriate event masks in the constructor. Enabling more than one event requires OR'ing corresponding event masks. These event masks are defined as constants in java.awt.AWTEvent.
- If the component wants to also notify the registered listeners for the event, then the overriding methods should call the parent version of the methods explicitly.
- Component class has a method processMouseMotionEvent, even though there is no event called MouseMotionEvent.

Steps for handling events using listeners or by the same component

Delegating to listeners	Handling own events (explicit enabling)
1. Create a listener class, either by implementing an event listener interface or extending an event adapter class. 2. Create an instance of the component 3. Create an instance of the listener class 4. Call addXXXListener on the component passing the listener object. (This step automatically enables the processing of this type of event. Default behavior of processEvent method in the component is to delegate the processing to processXXXEvent and that method will invoke appropriate listener class methods.)	1. Create a subclass of a component 2. Call enableEvents(XXX_EVENT_MASK) in the constructor. 3. Provide processXXXEvent and/or processEvent in the subclass component. If also want to notify the listeners, call parent method. 4. Create an instance of the subclass component

Chapter 13 Painting

- Some objects have default appearance. When they are created, OS decorates with a pre-defined appearance.
- Some components don't have any intrinsic appearance. These are Applet, Panel, Frame and Canvas. For these objects paint() method is used to render them.
  public void paint(Graphics g)
- The paint() method provides a graphics context (an instance of Graphics class) for drawing. This is passed as a method argument.
- A Graphics object encapsulates state information needed for the basic rendering operations that Java supports. This state information includes the following properties:
  - The Component object on which to draw.
  - A translation origin for rendering and clipping coordinates.
  - The current clip.
  - The current color.
  - The current font.
  - The current logical pixel operation function (XOR or Paint).
  - The current XOR alternation color.
- We can use this Graphics object to achieve the following functionality:
  1. Selecting a color – g.setColor(Color)
     There are 13 predefined colors in Color class. Or create a new color using Color(R,G,B)
  2. Selecting a Font – g.setFont(Font)
     A Font is created by Font(String name, int style, int size)
  3. Drawing and Filling – Various draw, fill methods
  4. Clipping – g.setClip(Shape) or g.setClip(x, y, width, height)
- Graphics class is an abstract class. It cannot be created. But an instance can be obtained in 2 ways.
  1. Every component has an associated graphics context. Get this using getGraphics method.
  2. Given an existing Graphics object, call create() on that object to create a new one.
  In both cases, after its use call dispose method on Graphics, to free the resources. We shouldn't call dispose on the graphics context passed into paint() method, since it's just temporarily made available.
- JVM calls paint() spontaneously under 4 circumstances
  1. After exposure
  2. After de-iconification
  3. Shortly after init returns (Applets only)
  4. Browser returns to a page contains the Applet (Applets only)
- In all cases, clip region is set appropriately. If only a small portion is exposed, no time is wasted in drawing already drawn pixels.
- Programs can also call paint(). But normally they achieve this by calling repaint(). Repaint() schedules a call to update() method (every 100 ms in most platforms). This is to ensure that JVM is never overwhelmed with the events.
- update() restores the component's background color and calls paint(). If you don't want to erase the previously drawn content, override update() and just call paint() from it. (A common practice).
- Event handlers that need to modify the screen according to input events, usually store the state information in instance variables and call repaint().
- Images can be created from empty (using createImage(int width, int height) method) or loaded from external image files (using getImage() method in Toolkit class). Then they can be modified using the graphics context associated with the image. They can be drawn on the component using the drawImage method of the Graphics context of the component.

Chapter 14 Applets and HTML

- <APPLET> tag specifies applet information in a HTML page
- It must be enclosed between <BODY> and </BODY>
- </APPLET> tag is mandatory to close the <APPLET> tag
- CODE, WIDTH and HEIGHT are mandatory attributes within <APPLET> tag. Their order is not significant. Instead of the class file, applet can specified via a serialized file using OBJECT attribute. If we specify OBJECT attribute CODE attribute is not specified and vice versa.
- In HTML 4.0, OBJECT tag can be specified instead of APPLET tag. With OBJECT tag, we specify the applet with CLASSID attribute.
- The following are other optional tags

Tag Name	Purpose
CODEBASE	Directory for the applet's class
ALT	Alternate text for browsers with no support for applets but can understand <APPLET> tag
HSPACE	Left/Right boundaries b/w other HTML elements on a page
VSPACE	Top/Bottom boundaries b/w other HTML elements on a page
ALIGN	Alignment with respect to other HTML elements
NAME	Name of the applet for inter-applet communication
ARCHIVE	Name of the JAR file (Lot of files can be downloaded in a single download to reduce the time needed) Even multiple jar files can be specified, separated by commas.
OBJECT	Specified if CODE attribute is not present and vice versa. Applet is read in from the specified serialized file.

- Between <APPLET> and </APPLET>, PARAM tags can be specified. These are used to pass parameters from HTML page to the applet.
- <PARAM NAME = "name" VALUE = "value">
- Applets call getParameter(name) to get the parameter. The name is not case sensitive here.
- The value returned by getParameter is case sensitive, it is returned as defined in the HTML page.
- If not defined, getParameter returns null.
- Text specified between <APPLET> and </APPLET> is displayed by completely applet ignorant browsers, who cannot understand even the <APPLET> tag.
- If the applet class has only non-default constructors, applet viewer throws runtime errors while loading the applet since the default constructor is not provided by the JVM. But IE doesn't have this problem. But with applets always do the initialization in the init method. That's the normal practice.
- Methods involved in applet's lifecycle.

Method	Description
void init()	This method is called only once by the applet context to inform the applet that it has been loaded into the system. Always followed by calls to start() and paint() methods. Same purpose as a constructor. Use this method to perform any initialization.
void start()	Applet context calls this method for the first time after calling init(), and thereafter every time the applet page is made visible.
void stop()	Applet context calls this method when it wants the applet to stop the execution. This method is called when the applet page is no longer visible.
void destroy()	This method is called to inform the applet that it should relinquish any system resources that it had allocated. Stop() method is called prior to this method.
void paint(Graphics g)	Applets normally put all the rendering operations in this method.

- Limitations for Applets:
  - Reading, writing or deleting files on local host is not allowed.
  - Running other applications from within the applet is prohibited.
  - Calling System.exit() to terminate the applet is not allowed.
  - Accessing user, file and system information, other than locale-specific information like Java version, OS name and version, text-encoding standard, file-path and line separators, is prohibited.
  - Connecting to hosts other than the one from which the applet was loaded is not permitted.
  - Top-level windows that an applet creates have a warning message for applets loaded over the net.
- Some other methods of Applet class

Method	Description
URL getDocumentBase()	Returns the document URL, i.e. the URL of the HTML file in which the applet is embedded.
URL getCodeBase()	Returns the base URL, i.e. the URL of the applet class file that contains the applet.
void showStatus(String msg)	Applet can request the applet context to display messages in its "status window".

Chapter 15 I/O

- Inside JVM, text is represented in 16 bit Unicode. For I/O, UTF (**U**CS (Universal Character set) **T**ransformation **F**ormat) is used. UTF uses as many bits as needed to encode a character.
- Often programs need to bring in information from an external source or send out information to an external destination. The information can be anywhere: in a file, on disk, somewhere on the network, in memory, or in another program. Also, it can be of any type: objects, characters, images, or sounds.
- To bring in information, a program opens a stream on an information source (a file, memory or a socket) and reads the information serially. Similarly, a program can send information to an external destination by opening a stream to a destination and writing the information out serially.
- No matter where the information is coming from or going to and no matter what type of data is being read or written, the algorithms for reading and writing data is pretty much always the same.

Reading	**Writing**
open a stream	open a stream
while more information	while more information
read information	write information
close the stream	close the stream

- For this kind of general I/O Stream/Reader/Writer model is used. These classes are in java.io package. They view the input/output as an ordered sequence of bytes/characters.
- We can create I/O chains of arbitrary length by chaining these classes.
- These classes are divided into two class hierarchies based on the data type (either characters or bytes) on which they operate. Streams operate on bytes while Readers/Writers operate on chars.
- However, it's often more convenient to group the classes based on their purpose rather than on the data type they read and write. Thus, we can cross-group the streams by whether they read from and write to data "sinks" (Low level streams) or process the information as its being read or written (High level filter streams).

Low Level Streams / Data sink streams

- Low Level Streams/Data sink streams read from or write to specialized data sinks such as strings, files, or pipes. Typically, for each reader or input stream intended to read from a specific kind of input source, java.io contains a parallel writer or output stream that can create it. The following table gives java.io's data sink streams.

Sink Type	Character Streams	Byte Streams	Purpose
Memory	CharArrayReader, CharArrayWriter	ByteArrayInputStream, ByteArrayOutputStream	Use these streams to read from and write to memory. You create these streams on an existing array and then use the read and write methods to read from or write to the array.
Memory	StringReader, StringWriter	StringBufferInputStream	Use StringReader to read characters from a String as it lives in memory. Use StringWriter to write to a String. StringWriter collects the characters written to it in a StringBuffer, which can then be converted to a String. StringBufferInputStream is similar to StringReader, except that it reads bytes from a StringBuffer.
Pipe	PipedReader, PipedWriter	PipedInputStream, PipedOutputStream	Implement the input and output components of a pipe. Pipes are used to channel the output from one program (or thread) into the input of another.
File	FileReader, FileWriter	FileInputStream, FileOutputStream	Collectively called file streams, these streams are used to read from or write to a file on the native file system.

High Level Filter Streams / Processing streams

- Processing streams perform some sort of operation, such as buffering or character encoding, as they read and write. Like the data sink streams, java.io often contains pairs of streams: one that performs a particular operation during reading and another that performs the same operation (or reverses it) during writing. This table gives java.io's processing streams.

Process	Character Streams	Byte Streams	Purpose
Buffering	BufferedReader, BufferedWriter	BufferedInputStream, BufferedOutputStream	Buffer data while reading or writing, thereby reducing the number of accesses required on the original data source. Buffered streams are typically more efficient than similar nonbuffered streams.
Filtering	FilterReader, FilterWriter	FilterInputStream, FilterOutputStream	Abstract classes, like their parents. They define the interface for filter streams, which filter data as it's being read or written.
Converting between Bytes and Characters	InputStreamReader, OutputStreamWriter	N/A	A reader and writer pair that forms the bridge between byte streams and character streams. An InputStreamReader reads bytes from an InputStream and converts them to characters using either the default character-encoding or a character-encoding specified by name. Similarly, an OutputStreamWriter converts characters to bytes using either the default character-encoding or a character-encoding specified by name and then writes those bytes to an OutputStream.
Concatenation	N/A	SequenceInputStream	Concatenates multiple input streams into one input stream.
Object Serialization	N/A	ObjectInputStream, ObjectOutputStream	Used to serialize objects.
Data Conversion	N/A	DataInputStream, DataOutputStream	Read or write primitive Java data types in a machine-independent format. Implement DataInput/DataOutput interfaces.
Counting	LineNumberReader	LineNumberInputStream	Keeps track of line numbers while reading.
Peeking Ahead	PushbackReader	PushbackInputStream	Two input streams each with a 1-character (or byte) pushback buffer. Sometimes, when reading data from a stream, you will find it useful to peek at the next item in the stream in order to decide what to do next. However, if you do peek ahead, you'll need to put the item back so that it can be read again and processed normally. Certain kinds of parsers need this functionality.
Printing	PrintWriter	PrintStream	Contain convenient printing methods. These are the easiest streams to write to, so you will often see other writable streams wrapped in one of these.

- Reader and InputStream define similar APIs but for different data types. For example, Reader contains these methods for reading characters and arrays of characters:
  abstract int read() throws IOException
  int read(char cbuf[]) throws IOException

int read(char cbuf[], int offset, int length) throws IOException
InputStream defines the same methods but for reading bytes and arrays of bytes:
abstract int read() throws IOException
int read(byte cbuf[]) throws IOException
int read(byte cbuf[], int offset, int length) throws IOException
Also, both Reader and InputStream provide methods for marking a location in the stream, skipping input, and resetting the current position.

Both Reader and InputStream are abstract. Subclasses should provide implementation for the read() method.

- Writer and OutputStream are similarly parallel. Writer defines these methods for writing characters and arrays of characters:
abstract int write(int c) throws IOException
int write(char cbuf[])throws IOException
int write(char cbuf[], int offset, int length) throws IOException
And OutputStream defines the same methods but for bytes:
abstract int write(int c) throws IOException
int write(byte cbuf[]) throws IOException
int write(byte cbuf[], int offset, int length) throws IOException

Writer defines extra methods to write strings.
void write(String str) throws IOException
void write(String str, int offset, int length) throws IOException

Both Writer and OutputStream are abstract. Subclasses should provide implementation for the write() method.

Constructors for some common streams, reader and writers:

FileInputStream(String name) throws FileNotFoundException
FileInputStream(File file) throws FileNotFoundException
FileInputStream(FileDescriptor fdObj)

FileOutputStream(String name) throws FileNotFoundException
FileOutputStream(String name, boolean append) throws FileNotFoundException
FileOutputStream (File file) throws FileNotFoundException
FileOutputStream (FileDescriptor fdObj)

DataInputStream(InputStream in)
DataOutputStream(OutputStream out)

BufferedInputStream(InputStream in)
BufferedInputStream(InputStream in, int size)
BufferedOutputStream(OutputStream out)
BufferedOutputStream(OutputStream out, int size)

FileReader(File file) throws FileNotFoundException
FileReader (FileDescriptor fdObj)
FileReader (String name) throws FileNotFoundException

FileWriter(File file) throws IOException
FileWriter(String name) throws IOException
FileWriter(String name, boolean append) throws IOException
FileWriter(FileDescriptor fdObj)

InputStreamReader(InputStream in)
InputStreamReader(InputStream in, String encodingName) throws
UnsupportedEncodingException

OutputStreamWriter(OutputStream out)
OutputStreamWriter (OutputStream out, String encodingName) throws
UnsupportedEncodingException

PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoflush)
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoflush)

BufferedReader(Reader in)
BufferedReader(Reader in, int size)
BufferedWriter(Writer out)
BufferedWriter (Writer out, int size)

Encoding Name	Character Set Name
8859_1	ISO Latin-1 (subsumes ASCII)
8859_2	ISO Latin-2
8859_3	ISO Latin-3
8859_4	ISO Latin / Cyrillic
UTF8	Standard UTF-8 (subsumes ASCII)

- OutputStreamWriter and InputStreamReader are the only ones where you can specify an encoding scheme apart from the default encoding scheme of the host system. getEncoding method can be used to obtain the encoding scheme used.
- With UTF-8 Normal ASCII characters are given 1 byte. All Java characters can be encoded with at most 3 bytes, never more.
- All of the streams--readers, writers, input streams, and output streams--are automatically opened when created. You can close any stream explicitly by calling its close method. Or the garbage collector can implicitly close it, which occurs when the object is no longer referenced.
- Closing the streams automatically flushes them. You can also call flush method.
- New FileWriter("filename") or FileOutputStream("filename") will overwrite if "filename" is existing or create a new file, if not existing. But we can specify the append mode in the second argument.
- Print writers provide the ability to write textual representations of Java primitive values. They have to be chained to the low-level streams or writers. Methods in this class never throw an IOException.
- PrintStream and PrintWriter classes can be created with autoflush feature, so that each println method will automatically be written to the next available stream. PrintStream is deprecated.(though System.out and System.err are still of this type)
- System.in is of InputStream type.
- System.in, System.out, System.err are automatically created for a program, by JVM.
- Use buffered streams for improving performance.BufferedReader provides readLine method.

- User defined classes must implement Serializable or Externalizable interfaces to be serialized.
- Serializable is a marker interface with no methods. Externalizable has two methods to be implemented – readExternal(ObjectInput) and writeExternal(ObjectOutput).
- ObjectOutputStream can write both Java Primitives and Object hierarchies. When a compound object is serialized all its constituent objects that are serializable are also serialized.
- ObjectOutputStream implements ObjectOutput, which inherits from DataOutput.

- All AWT components implement Serializable (Since Component class implements it), so by default we can just use an ObjectOutputStream to serialize any AWT component.

- File class is used to navigate the file system.
- Constructing a File instance (or Garbage-Collecting it) never affects the file system.
- File class doesn't have a method to change the current working directory.
- File constructors

Constructor	Description
File(File dir, String name)	Creates a File instance that represents the file with the specified name in the specified directory
File(String path)	Creates a File instance that represents the file whose pathname is the given path argument.
File(String path, String name)	Creates a File instance whose pathname is the pathname of the specified directory, followed by the separator character, followed by the name argument.

- File methods

Method	Description
boolean canRead()	Tests if the application can read from the specified file.
boolean canWrite()	Tests if the application can write to this file.
boolean delete()	Deletes the file specified by this object.
boolean exists()	Tests if this File exists.
String getAbsolutePath()	Returns the absolute pathname of the file represented by this object.
String getCanonicalPath()	Returns the canonical form of this File object's pathname. **..** and **.** are resolved.
String getName()	Returns the name of the file represented by this object.
String getParent()	Returns the parent part of the pathname of this File object, or null if the name has no parent part.
String getPath()	Returns the pathname of the file represented by this object.
boolean isAbsolute()	Tests if the file represented by this File object is an absolute pathname.
boolean isDirectory()	Tests if the file represented by this File object is a directory.
boolean isFile()	Tests if the file represented by this File object is a "normal" file.
long lastModified()	Returns the time that the file represented by this File object was last modified.
long length()	Returns the length of the file (in bytes) represented by this File object.
String[] list()	Returns a list of the files in the directory specified by this File object.
String[] list(FilenameFilter)	Returns a list of the files in the directory specified by this File that satisfy the specified filter. FileNameFilter is an interface that has a method accept(). This list method will call accept for each entry in the list of files and only returns the files for which accept returns true.
boolean mkdir()	Creates a directory whose pathname is specified by this File object.
boolean mkdirs()	Creates a directory whose pathname is specified by this File object, including any necessary parent directories.
boolean renameTo(File)	Renames the file specified by this File object to have the pathname given by the File argument.

- Instances of the file descriptor class serve as an opaque handle to the underlying machine-specific structure representing an open file or an open socket.
- Applications should not create their own file descriptors

- RandomAccessFile lets you read/write at arbitrary places within files.
- RAF provides methods to read/write bytes.
- RAF also provides methods to read/write Java primitives and UTF strings. (RAF implements the interfaces DataInput and DataOutput)
- File and RAF instances should be closed when no longer needed.
- All reading/writing operations throw an IOException. Need to catch or declare our methods to be throwing that exception.
- Read/Write methods throw a SecurityException if the application doesn't have rights for the file.
- RAF cannot be chained with streams/readers/writers.

- RAF Constructors

Constructor	Description
RandomAccessFile(File file, String mode) throws FileNotFoundException, IllegalArgumentException, SecurityException	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
RandomAccessFile(String name, String mode) throws FileNotFoundException, IllegalArgumentException, SecurityException	Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

- The mode argument must either be equal to "r" or "rw", indicating either to open the file for input or for both input and output.

- Some RAF methods

Method	Description
long getFilePointer() throws IOException	Returns the offset from the beginning of the file, in bytes, at which the next read or write occurs.
void seek(long pos) throws IOException	Sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs. The offset may be set beyond the end of the file. Setting the offset beyond the end of the file does not change the file length. The file length will change only by writing after the offset has been set beyond the end of the file.
long length() throws IOException	Returns the length of this file, measured in bytes.