nexwave»

Nexwave

Java & J2EE

Notes

nexwave»

Contents

Language Fundamentals

A Java program is mostly a collection of objects talking to other objects by invoking each other's methods. Every object is of a certain type, and that type is defined by a class or an interface. Most Java programs use a collection of objects of many different types.

Bytecode: Java makes platform independence possible by translating a program into bytecode instead of machine code. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system or JVM (Java Virtual Machine). Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments, because only the JVM needs to be implemented from each platform. Once the run-time package exists for a given system, any Java program can run on it. Although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs

Class: A template that describes the kinds of state and behavior that objects of its type support.

Object: At runtime, when the Java Virtual Machine (JVM) encounters the new keyword, it will use the appropriate class to make an object which is an instance of that class. That object will have its own state, and access to all of the behaviors defined by its class.

State (instance variables): Each object (instance of a class) will have its own unique set of instance variables as defined in the class. Collectively, the values assigned to an object's instance variables make up the object's state.

Behavior (methods): When a programmer creates a class, she creates methods for that class. Methods are where the class' logic is stored. Methods are where the real work gets done. They are where algorithms get executed, and data gets manipulated.

Identifiers and Keywords

All the Java components we just talked about—classes, variables, and methods— need names. In Java these names are called identifiers, and, as you might expect, there are rules for what constitutes a legal Java identifier. Beyond what's legal, though, Java programmers (and Sun) have created conventions for naming methods, variables, and classes.

Like all programming languages, Java has a set of built-in keywords. These keywords must not be used as identifiers.

abstract	**double**	**int**	**strictfp**
boolean	**else**	**interface**	**super**
break	**extends**	**long**	**switch**
byte	**final**	**native**	**synchronized**
case	**finally**	**new**	**this**
catch	**float**	**package**	**throw**
char	**for**	**private**	**throws**
class	**goto**	**protected**	**transient**
const	**if**	**public**	**try**
continue	**implements**	**return**	**void**
default	**import**	**short**	**volatile**
do	**instanceOf**	**static**	**while**

Apart from these 48 keywords, there are three more words : **null, true and false**. Many authors list these words as keywords too. But, Java Language Specification says that these are not keywords but literals.

Inheritance
Central to Java and other object-oriented (OO) languages is the concept of inheritance, which allows code defined in one class to be reused in other classes. In Java, you can define a general (more abstract) superclass, and then extend it with more specific subclasses. The superclass knows nothing of the classes that inherit from it, but all of the subclasses that inherit from the superclass must explicitly declare the inheritance relationship. A subclass that inherits from a superclass is automatically given accessible instance variables and methods defined by the superclass, but is also free to override superclass methods to define more specific behavior. For example, a Car superclass class could define general methods common to all automobiles, but a Ferrari subclass could override the accelerate() method.

Interfaces
A powerful companion to inheritance is the use of interfaces. Interfaces are like a 100-percent abstract superclass that defines the methods a subclass must support, but not how they must be supported. In other words, an Animal interface might declare that all Animal implementation classes have an eat() method, but the Animal interface doesn't supply any logic for the eat() method. That means it's up to the classes that implement the Animal interface to define the actual code for how that particular Animal type behaves when its eat() method is invoked.

Differences between Interface and Abstract class?

Abstract Class	Interfaces
An abstract class can provide complete, default code and/or just the details that have to be overridden.	An interface cannot provide any code at all,just the signature.
In case of abstract class, a class may extend only one abstract class.	A Class may implement several interfaces.
An abstract class can have non-abstract methods.	All methods of an Interface are abstract.
An abstract class can have instance variables.	An Interface cannot have instance variables.
An abstract class can have any visibility: public, private, protected.	An Interface visibility must be public (or) none.
If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly.	If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method.
An abstract class can contain constructors .	An Interface cannot contain constructors .
Abstract classes are fast.	Interfaces are slow as it requires extra indirection to find corresponding method in the actual class.

Finding Other Classes
As we'll see later in the book, it's a good idea to make your classes cohesive. That means that every class should have a focused set of responsibilities. Even a simple Java program uses objects from many different classes: some that you created, and some built by others (such as Sun's Java API classes). Java organizes classes into packages, and uses import statements to give programmers a consistent way to manage naming of, and access to, classes they need. The exam covers a lot of concepts related to packages and class access.

Legal Identifiers
Technically, legal identifiers must be composed of only Unicode characters, numbers, currency symbols, and connecting characters (like underscores. Here are the rules you do need to know:

- Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore (_). Identifiers cannot start with a number!
- After the first character, identifiers can contain any combination of letters, currency characters, connecting characters, or numbers.
- In practice, there is no limit to the number of characters an identifier can contain.
- You can't use a Java keyword as an identifier.

- Identifiers in Java are case-sensitive; foo and FOO are two different identifiers.

Examples of legal and illegal identifiers follow, first some legal identifiers:

```
int _a;
int $c;
int ______2_w;
int _$;
int this_is_a_very_detailed_name_for_an_identifier;
```

The following are illegal (it's your job to recognize why):

```
int :b;
int -d;
int e#;
int .f;
int 7g;
```

Important Points to Remember

1. Source file's elements (should be in this order)
   - Package declaration
   - Import statements
   - 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. J 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.
    a. **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.

    b. **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. Specify else part or initialize the local variable explicitly.

```
public String testMethod ( int a) {
String tmp;
if ( a > 0 ) tmp = "Positive";
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
    - ✓ Declared. (int[] a; String b[]; Object []c; Size should not be specified now)
    - ✓ Allocated (constructed). (a = new int[10]; c = new String[arraysize])
    - ✓ 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.

2 ways to explicitly initialize arrays:

1)

```
for (int i = 0; i < iA.length; i++) iA[i] = 5;
for (int i = 0; i < sA.length; i++) sA[i] = new Student();
```

2) (Initialization the time of allocation.)

```
iA = new int[] {5, 5, 5, 5, 5, 5, 5, 5, 5, 5 };
sA = new Student[] { new Student(), new Student(), new Student(), new Student(), new Student() };
```

The above statements are creating an int array of 10 elements and a Student array of 5 elements respectively. Note that, the size is not specified explicitly.

Multidimensional Arrays:

- ✓ Java supports Arrays of Arrays. This means you can have something like:
  int[][] iAA; iAA is an array. And each of it's elements points to an object which is an array of ints.

It can be instantiated like this:

- ✓ int[][] iAA = new int[2][3]; ie. An array containing 2 elements. Each element in turn points to another array of ints containing 3 elements.
- ✓ Rules for multidimensional array are exactly same as single dimensional array. Eg. in the above case, the 2 places of iAA are pointing to null.
  Or iAA = new int[][]{ {1, 2, 3}, null }; Here, first element of iAA is pointing to an array containing {1, 2, 3} and the second element is pointing to null.

Array features:

Array size is always maintained with the array object as a final instance variable named 'length'. Ex. iAA.length . You cannot assign any value to it. ie. you cannot do iAA.length = 4;

Array Pitfalls:

An Array is a first class java object. Eg. Assuming int[] iA == new int[3]; (iA instance of Object) is true.

When you create an array of primitive type eg. int[] iA == new int[3]; , you create 1 object and three primitives initialized to 0. But when you create an array of any java object, eg. String[] sA == new String[3]; , you create 1 object and 3 places initially pointing to 'null' to hold 3 string objects.

Difference between

- ✓ int[] a, b; // a and b both are arrays
- ✓ and int a[], b; // a is an array but b is just an int.

Indexing starts with 0. ie. First element is accessed as: iA[0] = 10;

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)

(i) All Java applications begin execution by calling main ()
(ii) When a class member is defined as public. Then that member may be accessed by code outside the class in which it is declared.
(iii) The opposite of public is private which prevents a member from being used by code defined outside of its class.
(iv) The keyword static allows main() to be called without having to instantiate a particular instance of the class. This is mandatory because main () is called by the Java interpreter before any objects are made.
(v) CASE SENSITIVE: Main () is different from main(). It is important to know that that Main() would be compiled. But the Java interpreter would report an error if it would not find main().

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.

Garbage collection is one of the most important feature of Java, Garbage collection is also called automatic memory management as JVM automatically removes the unused variables/objects (value is null) from the memory. User program can't directly free the object from memory, instead it is the job of the garbage collector to automatically free the objects that are no longer referenced by a program. Every class inherits finalize() method from java.lang.Object, the finalize() method is called by garbage collector when it determines no more references to the object exists. In Java, it is good idea to explicitly assign null into a variable when no more in use.

calling System.gc() and Runtime.gc(), JVM tries to recycle the unused objects, but there is no guarantee when all the objects will garbage collected.

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.

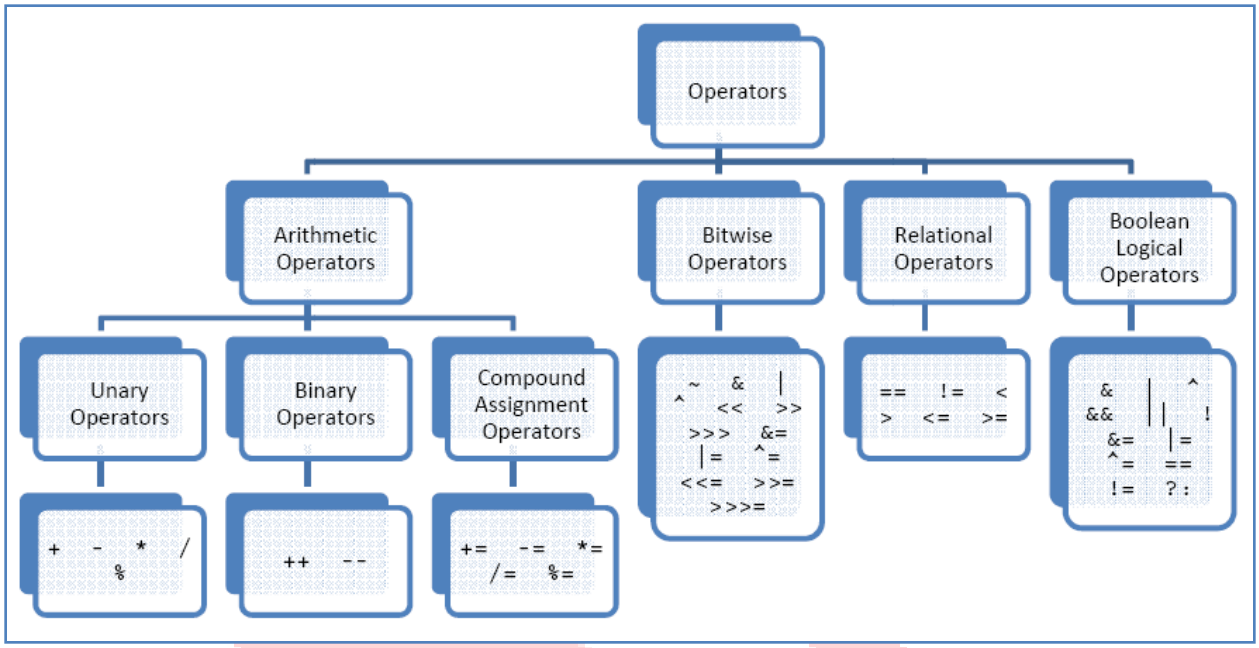
56. finalize can be called explicitly, but it does not garbage collect the object.

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.

Operators & 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 modified after the value is taken for the execution of statement. In prefix notation, value of the variable/ expression is modified before 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.
- ✓ % - 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 required since 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 gets 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

Operator Precedence Chart

Highest			
()	[]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

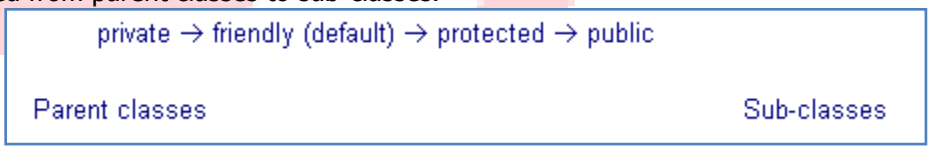
Modifiers

1. Modifiers are Java keywords that provide information to compiler about the nature of the code, data and classes.

2. **Access specifiers:** Java offers four access specifiers, listed below in decreasing accessibility:
   1. **Public**- *public* classes, methods, and fields can be accessed from everywhere.
   2. **Protected**- *protected* methods and fields can only be accessed within the same class to which the methods and fields belong, within its subclasses, and within classes of the same package.
   3. **Default(no specifier)-** If you do not set access to specific level, then such a class, method, or field will be accessible from inside the same package to which the class, method, or field belongs, but not from outside this package.
   4. **Private**- *private* methods and fields can only be accessible within the same class to which the methods and fields belong. *private* methods and fields are not visible within subclasses and are not inherited by subclasses.

Situation	public	protected	default	private
Accessible to class from same package?	yes	yes	yes	no
Accessible to class from different package?	yes	no, *unless it is a subclass*	no	no

- ✓ 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 classes can access the features. An instance of a class can access the private features of another instance of the same class.
- ✓ '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.



3. final

- ✓ final features cannot be changed.
- ✓ The final modifier applies to classes, methods, and variables.
- ✓ final classes cannot be sub-classed.
- ✓ You can declare a variable in any scope to be final.
- ✓ You may, if necessary, defer initialization of a final local variable. Simply declare the local variable and initialize it later (for final instance variables. You must initialize them at the time of declaration or in constructor).

- ✓ final variables cannot be changed (result in a compile-time error if you do so)
- ✓ 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.
- ✓ A blank final is a final variable whose declaration lacks an initializer.
- ✓ 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.
- ✓ If a final variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object.
- ✓ This applies also to arrays, because arrays are objects; if a final variable holds a reference to an array, then the components of the array may be changed by operations on the array, but the variable will always refer to the same array
- ✓ A blank final instance variable must be definitely assigned at the end of every constructor of the class in which it is declared; otherwise a compile-time error occurs.
- ✓ A class can be declared final if its definition is complete and no subclasses are desired or required.
- ✓ A compile-time error occurs if the name of a final class appears in the extends clause of another class declaration; this implies that a final class cannot have any subclasses.
- ✓ A compile-time error occurs if a class is declared both final and abstract, because the implementation of such a class could never be completed.
- ✓ Because a final class never has any subclasses, the methods of a final class are never overridden

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,
  - o if it has any abstract methods.
  - o if it doesn't provide implementation to any of the abstract methods it inherited
  - o 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.
- ✓ 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

Converting & Casting

Unary Numeric Promotion

- ✓ 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

- ✓ 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. (but can be cast)
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.

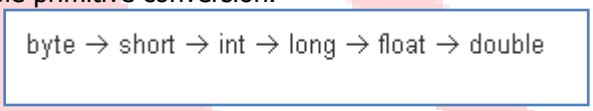
```
char c = 3;
// this will compile, even though a numeric literal is by default an
int since the range of char will accept the value
int a = 3;
char d = a; // this won't compile, since we're assigning an int to char
char e = -1; // this also won't compile, since the value is not in the
range of char
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.



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

Casting

Implicit cast
When you are widening a conversion: from a byte to an int

Explicit cast
When you are narrowing a conversion: from a double to a float

- Literal integer (e.g. 7) is implicitly a int, cast is done by the compiler
example: char d = 27;

- Adding two bytes can't be assigned to a byte without a cast. The result of a calculation with operands of type smaller then int will be promoted to an int, that is why the cast is necessary.
byte a = 10;
byte b = 2;
byte c = **(byte)** (a + b); // you have to put the explicit cast

c+=6; // This is possible without a cast
c=200; // Although 200 is an implicit int, you don't need a cast

Scope of variables

- ✓ Static variables have the longest scope; they are created when the class is loaded, and they survive as long as the class stays loaded in the JVM
- ✓ Instance variables are the next most long-lived; they are created when a new instance is created, and they live until the instance is removed
- ✓ Local variables are the next; they live as long as the method remains on the stack
- ✓ Block variables live only as long as the code block is executing

Most common scoping errors

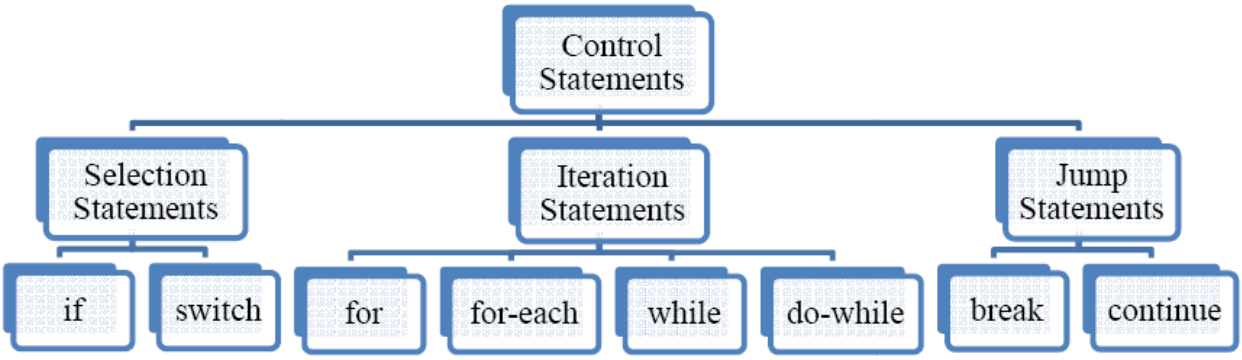
- ✓ Attempting to access a instance variable from a static context (typically main())
- ✓ Attempting to access a local variable from a nested method
- ✓ Attempting to access a block variable after the code block has completed

Flow Control

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 at least once.
- ✓ 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 or 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.

Flow Control:

{{}} is a valid statement block

if - else

condition statement can contain

- ✓ method calls
- ✓ only expr which evaluates to boolean value can be used as condition

if(false); else; // is legal

switch

- ✓ control falls through next statement unless appropriate action is taken
- ✓ all labels are optional
- ✓ at most one default label
- ✓ labels can be specified in any order
- ✓ type of the expression must be char, byte, short or int [cannot be boolean /long /floating pt] the type of case labels must be assignable to the type of switch expr
- ✓ can be nested & labels can be redefined in the nested blocks

for

- ✓ all expressions in header are optional (two semicolons are mandatory) for(;;) is infinite loop
- ✓ multiple variables can be given but must be of the SAME type

do | while

- ✓ condition must evaluate to boolean value
- ✓ while(true) is an infinite loop

break

- ✓ transfer control out of the current context (closest enclosing block)
- ✓ not possible to break out of if statement
  e.g. if(true) {break; }
- ✓ but if it is placed inside a labeled block, switch or loop usage of 'break' in if is valid.
- ✓ for loop : out of the loop body, terminating the loop & going to next statement after loop
- ✓ break outer - comes out of the loop labeled as 'outer' e.g.
  block: { break block; } //valid

continue

- ✓ can be used with for , while, do-while loop
- ✓ the rest of the loop is skipped with the execution continuing with the <incr expr>
  block: {break continue;} //is invalid

Objects & Classes

Object Characteristics: Three properties characterize objects:

1. Identity: the property of an object that distinguishes it from other objects.
2. State: describes the data stored in the object.
3. Behavior: describes the methods in the object's interface by which the object can be used.

Differentiate among instance variable, class variable and local variable.

- ✓ **Instance Variables** (Non-Static Fields): An instance variable is any field declared without the static modifier. It is called such because its value is unique to each instance of a class (or object).
- ✓ **Class Variables** (Static Fields): A class variable is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.
- ✓ **Local Variables** A local variable is a variable declared inside a method.

Source file declaration rules

- ✓ There can be only one public class per source code file
- ✓ Comments can appear at the beginning or end of any line in the source code file
- ✓ If there is a public class in a file, the name of the file must match the name of the public class.
- ✓ Package statement must be the first statement in the source code file.
- ✓ Import statement between the *package statement* and the *class declaration*
- ✓ Import & Package statement apply to all classes in the file
- ✓ A file can have more than one nonpublic class
- ✓ Files with no public classes can have a name that does not match any of the classes in the file

Reference Variables

- ✓ A reference variable can be of only one type, and once declared, can never be changed
- ✓ A reference is a variable, so it can be reassigned to different objects (unless declared final)
- ✓ A reference variable's type determines the methods that can be invoked on the object the variable is referencing (this is known at compile time)
- ✓ A reference variable can refer to any object of the same type as the declared reference, or it can refer to a subtype of the declared type (passing the IS-A test)
- ✓ A reference variable can be declared as a class type or as an interface type. If the reference variable is declared as an interface type, it can reference any object of any class that *implements* the interface (passing the IS-A test)

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.

Differences between method overloading and method overriding?

	Overloaded Method	Overridden Method
Arguments	Must change	Must not change
Return type	Can change	Can't change except for covariant returns
Exceptions	Can change	Can reduce or eliminate. Must not throw new or broader checked exceptions
Access	Can change	Must not make more restrictive (can be less restrictive)
Invocation	Reference type determines which overloaded version is selected. Happens at compile time.	Object type determines which method is selected. Happens at runtime.

Rules for overriding a method

- ✓ The overridden method has the same name.
- ✓ The argument list must exactly match (i.e. *int, long* is not the same as *long, int*) that of the overridden method. If the don't match, you end up with an overloaded method.
- ✓ The order of arguments is important
- ✓ The return type must be the same as, or a subtype of, the return type declared in the original overridden method in the super-class.
- ✓ The access level can't be more restrictive than the overridden method's
- ✓ The access level CAN be less restrictive than that of the overridden method

- ✓ Instance methods can be overridden only if they are inherited by the subclass. A subclass within the same package as the instance's super-class can override any super-class method that is not marked private or final. A subclass in a different package can override only those non-final methods marked public or protected (since protected methods are inherited by the subclass)
- ✓ Trying to override a private method is not possible because the method is not visible, that means that a *subclass can define a method with the same signature* without a compiler error!
- ✓ Trying to override a final method will give a compile error
- ✓ The overriding method CAN throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception
- ✓ The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method
- ✓ The overriding method can throw narrower or fewer exceptions.
- ✓ You cannot override a method marked final
- ✓ You cannot override a method marked static
- ✓ If a method is not visible it cannot be inherited.
- ✓ If a method cannot be inherited it cannot be overridden.
- ✓ An overriding method CAN be final

Overloaded methods

- ✓ Overloaded methods have the same name
- ✓ Overloaded methods must change the argument list
- ✓ Overloaded methods can change the return type
- ✓ Overloaded methods can change the access modifier
- ✓ Overloaded methods can declare new or broader checked exceptions

Which method is called

- ✓ Which overridden version of the method to call is decided at runtime based on the object type.
- ✓ Which overloaded version of the method to call is based on the reference type of the argument passed at compile 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 points.

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
// Compiler error, GrandParent method cannot be accessed
		//super.super.getWealth();
		// Calls Child method, due to dynamic method lookup
((STParent)this).getWealth();
// Calls Child method, due to dynamic method lookup
	((STGrandParent)this).getWealth();

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

Early Binding, Late Binding and Dynamic Binding

- ✓ Normally, Java resolves calls to methods dynamically, at run time. This is called late binding.
- ✓ However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.
- ✓ Dynamic binding or dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

Constructors

- ✓ Constructors have no return type i.e. void classname() is a method & not a constructor
- ✓ Constructors can be private, protected or public
- ✓ Private Constructors Suppose that you want to create a class but you don't want it to be instantiated as an object. Just declare the class's constructors as private.
- ✓ Constructors cannot be final, static or abstract.
- ✓ Constructors cannot be overridden but can be locally (same class) overloaded
- ✓ Local chaining of constructors is possible with the use of this reference

  e.g.

```
A(){this(0,true) } calls the appropriate constructor
A(int,boolean) {this(a,b,"X")} and so on
A(int,boolean,char) {……}
```

- ✓ super() is used in subclass to invoke constructors of the immediate superclass
- ✓ Java specifies that when using this() or super() it must occur as the first statement in a constructor (i.e both can not occur in the same constructor)
- ✓ Constructors can use any access modifier, including private
- ✓ The constructor name must match the name of the class
- ✓ Constructors must not have a return type
- ✓ It's legal to have a method with the same name as the class
- ✓ If you don't type a constructor into you class code, a default constructor will be automatically be generated by the compiler
- ✓ The default constructor is always a no-argument constructor
- ✓ If you want a no-argument constructor and you have typed any other constructor(s) into your class code, the compiler won't provide the no-argument constructor
- ✓ Every constructor has, as its first statement, either a call to an overloaded constructor (this()) or a call to the super-class constructor (super())
- ✓ If you create a constructor, and you do not have an explicit call to super() or an explicit call to this(), the compiler will insert a no-argument call to super(). (if there is no no-argument constructor in the super-class, a compile error will be generated).
- ✓ A call to super() can be either a no-argument call or can include arguments passed to the super constructor
- ✓ A no-argument constructor is not necessarily the default constructor, although the default constructor is always a no-argument constructor
- ✓ You cannot make a call to an instance method, or access an instance variable, until after the super constructor runs
- ✓ Only static variables and methods can be accessed as part of the call to super() or this().
- ✓ Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated
- ✓ Interfaces do not have constructors.
- ✓ The only way a constructor can be invoked is from within another constructor.

Differences between Contructors and Methods?

	Constructors	Methods
Purpose	Create an instance of a class	Group Java statements
Modifiers	Cannot be *abstract, final, native, static*, or *synchronized*	Can be *abstract, final, native, static*, or *synchronized*
Return Type	No return type, not even void	void or a valid return type
Name	Same name as the class (first letter is capitalized by convention) -- usually a noun	Any name except the class. Method names begin with a lowercase letter by convention -- usually the name of an action
this	Refers to another constructor in the same class. If used, it must be the first	Refers to an instance of the owning class. Cannot be used by static methods.

	line of the constructor	
super	Calls the constructor of the parent class. If used, must be the first line of the constructor	Calls an overridden method in the parent class
Inheritance	Constructors are not inherited	Methods are inherited

- ✓ if constructor does not have this() or super() then a super() call to the default constructor of super class is inserted (e.g. first constructor called is Object class) provided the subclass does not define non-default constructors in which case call super() constructor with right args e.g. super(0,true,"X") or it will look for default constructor in superclass

```
class A{
A(){
       this("1","2")
};
A(x,y) {
       this(x,y,"Z")
       };
A(x,y,z){…}
    }
class B extends A{
       B(string s) //will automatically call default constructor from a super()
       {
              print s;
       }
e.g
Mysub(int x, int y){
       super(num); count = x);
  Mysub(int x) {
       this(x,x);
```

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.

Difference between Class & Interface:

Class	Interface
Defines what states and behaviors an *object* can have.	Defines what methods a *class* can have.
Can extend another class or implement interfaces.	Can extend only another interface.
Can have methods with bodies defined as well as methods without bodies (i.e., abstract methods).	Cannot have any method with a defined body.
Can contain both instance variables and constants.	Can have only static final variables (i.e., constants).
Objects of a class can be instantiated.	Objects of an interface cannot be instantiated.

Inner Classes

✓ A "regular" inner class is declared inside the curly braces of another class, but outside any method or other code block.

- ✓ An inner class is a full-fledged member of the enclosing (outer) class, so it can be marked with an access modifier as well as the abstract or final modifiers. (Never both abstract and final together— remember that abstract must be subclassed, whereas final cannot be subclassed).
- ✓ An inner class instance shares a special relationship with an instance of the enclosing class. This relationship gives the inner class access to all of the outer class's members, including those marked private.
- ✓ To instantiate an inner class, you must have a reference to an instance of the outer class.
- ✓ From code within the enclosing class, you can instantiate the inner class using only the name of the inner class, as follows:

```
MyInner mi = new MyInner();
```

- ✓ From code outside the enclosing class's instance methods, you can instantiate the inner class only by using both the inner and outer class names, and a reference to the outer class as follows:

```
MyOuter mo = new MyOuter();
MyOuter.MyInner inner = mo.new MyInner();
```

- ✓ From code within the inner class, the keyword this holds a reference to the inner class instance. To reference the outer this (in other words, the instance of the outer class that this inner instance is tied to) precede the keyword this with the outer class name as follows: MyOuter.this;

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)

```
// 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 {
        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

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
             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();
  }
}
```

Difference between Class Methods and Instance Methods?

Class Methods	Instance Methods
Class methods are methods which are declared as static. The method can be called without creating an instance of the class	Instance methods on the other hand require an instance of the class to exist before they can be called, so an instance of a class needs to be created by using the new keyword. Instance methods operate on specific instances of classes.
Class methods can only operate on class members and not on instance members as class methods are unaware of instance members.	Instance methods of the class can also not be called from within a class method unless they are being called on an instance of that class.
Class methods are methods which are declared as static. The method can be called without creating an instance of the class.	Instance methods are not declared as static.

Anonymous Inner Classes

- ✓ Anonymous inner classes have no name, and their type must be either a subclass of the named type or an implementer of the named interface.
- ✓ An anonymous inner class is always created as part of a statement; don't forget to close the statement after the class definition with a curly brace. This is a rare case in Java, a curly brace followed by a semicolon.
- ✓ Because of polymorphism, the only methods you can call on an anonymous inner class reference are those defined in the reference variable class (or interface), even though the anonymous class is really a subclass or implementer of the reference variable type.
- ✓ An anonymous inner class can extend one subclass or implement one interface. Unlike non-anonymous classes (inner or otherwise), an anonymous inner class cannot do both. In other words, it cannot both extend a class and implement an interface, nor can it implement more than one interface.
- ✓ An argument-defined inner class is declared, defined, and automatically instantiated as part of a method invocation. The key to remember is that the class is being defined within a method argument, so the syntax will end the class definition with a curly brace, followed by a closing parenthesis to end the method call, followed by a semicolon to end the statement: });

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

Static Nested Classes

- ✓ Static nested classes are inner classes marked with the static modifier.
- ✓ A static nested class is not an inner class, it's a top-level nested class.
- ✓ Because the nested class is static, it does not share any special relationship with an instance of the outer class. In fact, you don't need an instance of the outer class to instantiate a static nested class.
- ✓ Instantiating a static nested class requires using both the outer and nested class names as follows:

```
BigOuter.Nested n = new BigOuter.Nested();
```

- ✓ A static nested class cannot access non-static members of the outer class, since it does not have an implicit reference to any outer instance (in other words, the nested class instance does not get an outer this reference).

Reference Card

	Top level nested class	Non static inner class	Local class	Anonymous class
Declaration Context	As static class member	As non-static class member	In block with non-static context	In block with non-static context
Accessibility Modifiers	All	All	None	None
Outer instance	No	Yes	Yes	Yes
Direct Access to enclosing context	Static members in enclosing context	All members in enclosing context	All members in enclosing context + local final variables	All members in enclosing context + local final variables
Defines static or non-static members	Both static and non-static	Only non-static	Only non-static	Only non-static
Constructors	Yes	Yes	Yes	No
Can use extends or implements clauses?	Yes	Yes	Yes	No

Garbage Collection

Java, they say, scores over other languages like C or C++ in memory management. In Java you need not worry about memory management since its garbage collection mechanism takes care of memory leaks. When an object cannot be referenced in a program or if it cannot be accessed anymore, maybe due to its reference being assigned to another object or its reference being set to null, it becomes eligible for garbage collection. The built in garbage collector mechanism then reclaims the memory which had been allocated to the object.

The storage allocated to an object is not recovered unless it is definitely no longer in use. Even though you may not be using an object any longer, you cannot say when, even if at all, it will be collected. Even

methods like System.gc() and Runtime.gc() cannot be relied upon in general, since some other thread might prevent the garbage collection thread from running.

An important consequence of the nature of automatic garbage collection is that there can still be memory leaks. If live, accessible references to unneeded objects are allowed to persist in a program, then those objects cannot be garbage collected. Therefore it is better to explicitly assign null to a variable when it is not needed any more. This is particularly important when implementing a collection.

Object lifetime

The lifetime of an object is from the time it is created to the time it is garbage collected. The finalization mechanism does provide a means for resurrecting an object after it is no longer in use and eligible for garbage collection, but finalization is rarely used for this purpose.

Cleaning up

Objects that are created and accessed by local references in a method are eligible for garbage collection when the method terminates, unless references to those objects are exported out of the method. This can occur if a reference is returned or thrown as an exception.

Object finalization

```
protected void finalize() throws Throwable;
```

A finalizer can be overridden in a method in a subclass to take appropriate action before the object is destroyed. A finalizer can catch and throw exceptions like other methods. However, any exception thrown but not caught by a finalizer when invoked by the garbage collector is ignored. The finalizer is only called once on an object, regardless of being interrupted by any exception during its execution. In case of finalization failures the object still remains eligible for garbage collection at the discretion of garbage collector unless it has been resurrected.

finalize () method:

Sometimes an object will need to perform some action when it is destroyed by the garbage collector. For example, if an object is holding some non java resources such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector. To add a finalizer to a class, you simply define a finalize() method. This method has the general form:

```
protected void finalize ();
```

You might find that the storage for an object never gets released because your program never nears the point of running out of storage. If your program completes and the garbage collector never gets around to releasing the storage for any of your objects, that storage will be returned to the operating system en masse as the program exits. This is a good thing, because garbage collection has some overhead, and if you never do it you never incur that expense.

Finalizers are guaranteed to be called before the memory used by an object is reclaimed. However there is no guarantee that any memory will ever be reclaimed. Hence there is no guarantee that finalize( ) will ever be called. There is a promise that barring catastrophic error conditions, all finalizers will be run on leftover objects when the Java virtual machine exits. However this is likely too late if your program is waiting for a file handle to be released. Besides it is not convincing that it happens anyway. Therefore it is vital that you never rely on a finalizer to free finite resource, such as file handles, that may be needed late by your program.

Exception Handling

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

How an exception is handled:

1. When an exception is occurred, JVM throws an object of that type of exception on the line where the exception occurred. If that line is inside a try-catch block, then a match for that exception is searched among the catch blocks.
2. If no match is found, it is checked whether this try-catch is a nested try-catch. If it is, then a match for the exception is searched among the parent try-catch block. But before going there, the finally block of the nested try-catch is executed.
3. If no match is found in the parent try-catch block, then the exception is thrown back on the line of the calling method. If that calling line is inside a try-catch block, the catch statements are checked for a match according to rules 1 and 2.
4. If no match is found, again the exception is thrown back on the line of the caller of this method. But before going there, the finally block of the current method is executed. This continues until the main method is reached.
5. When no match for the exception is found in the main method, the exception is thrown to JVM. But before going there, the finally block of the main method is executed. Now, JVM prints the exception and terminates the program.
6. If, in the middle of travelling, a valid return statement is found, then the exception is not thrown to the caller method, rather it is destroyed. Therefore, the current method returns and program execution continues from the next line of the calling method.

- ✓ If an exception occurs and an appropriate exception handler cannot be found, then the exception is handled by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exceptoin occurred, and terminates the program.
- ✓ When a catch handler throws an exception, a matching catch handler is searched in the outer try-catch block. If no match is found, then the default handler handles it.

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.

Checked & Unchecked Exceptions

- ✓ **Unchecked Exceptions**: The types of exceptions that need not be included in a methods **throws** list
  - `ArithmeticException`
  - `ArrayIndexOutOfBoundsException`
  - `ClassCastException`
  - `IndexOutOfBoundsException`
  - `IllegalStateException`
  - `NullPointerException`
  - `SecurityException`
- ✓ **Checked Exceptions:** The types of exceptions that must be included in a methods **throws** list if that method can generate one of these exceptions and does not handle it itself.
  - `ClassNotFoundException`
  - `CloneNotSupportedException`
  - `IllegalAccessException`
  - `InstantiationException`
  - `InterruptedException`
  - `NoSuchFieldException`
  - `NoSuchMethodException`

try – catch – finally

- ✓ A try can appear with no catch, but with just a finally.
- ✓ The *scope* of catch's variable is confined to the catch block itself.
- ✓ The *most general* exception goes *LAST* in a catch block.
- ✓ The compiler *won't allow you* to mistakenly put the most general exception first.
- ✓ The EXCEPTION e part of the catch statement must go in *parens*. ie. catch (IOEXCEPTION e) { }

finally

- ✓ A finally block is a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
- ✓ finally is (almost) always executed, even if there's no exception thrown.
- ✓ The only exception to the above rule is when System.exit(...); is called above the finally.

- ✓ finally is executed *just before* any return statement appearing above it. Then the return. So any return code on the return isn't sent out until after the finally.
- ✓ finally is executed *even if* an *outside method* called by the try block *throws an unknown or uncaught exception.*
- ✓ The finally block is useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.

Chained Exceptions

The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception. Lets take a simple example. You are trying to read a number from the disk and using it to divide a number. Think the method throws an ArithmeticException because of an attempt to divide by zero (number we got). However, the problem was that an I/O error occurred, which caused the divisor to be set improperly (set to zero). Although the method must certainly throw an ArithmeticException, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. This is the place where chained exceptions come in to picture.

```
Throwable getCause( )
Throwable initCause(Throwable causeExc)
```

Difference between throw and throws

throws: Used in a method's signature if a method is capable of causing an exception that it does not handle, so that callers of the method can guard themselves against that exception. If a method is declared as throwing a particular class of exceptions, then any other method that calls it must either have a try-catch clause to handle that exception or must be declared to throw that exception (or its superclass) itself.

A method that does not handle an exception it throws has to announce this:

```
public void myfunc(int arg) throws MyException {}
```

throw: Used to trigger an exception. The exception will be caught by the nearest try-catch clause that can catch that type of exception. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.

To throw an user-defined exception within a block, we use the throw command:

```
throw new MyException("I always wanted to throw an exception!");
```

Difference between Error and Exception

An **error** is an irrecoverable condition occurring at runtime. Such as OutOfMemory error. These JVM errors and you can not repair them at runtime.Though error can be caught in catch block but the execution of application will come to a halt and is not recoverable.

While **exceptions** are conditions that occur because of bad input etc. e.g. FileNotFoundException will be thrown if the specified file does not exist. Or a NullPointerException will take place if you try using a null reference. In most of the cases it is possible to recover from an exception (probably by giving user a feedback for entering proper values etc.)

Assertions

Programming With Assertions:

- An *assertion* is a statement in the Java™ programming language that enables you ***to test your assumptions*** about your program.
- Each assertion contains a boolean expression that you believe will be true when the assertion executes. If it is not true, the system will throw an error.
- By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors.
- assertions serve to document the inner workings of your program, enhancing maintainability.
- assert *Expression*$_1$;
    - where *Expression*$_1$ is a boolean expression.
    - When the system runs the assertion, it evaluates *Expression*$_1$ and if it is false throws an AssertionError with no detail message.
- assert *Expression*$_1$: *Expression*$_2$;
    - *Expression*$_1$ is a boolean expression.
    - *Expression*$_2$ is an expression that has a value.
    - *Expression2* **cannot be an invocation of a method that is declared *void***.
    - Use this version of the assert statement to provide a detail message for the AssertionError.
    - The system passes the value of *Expression*$_2$ to the appropriate AssertionError constructor, which uses the string representation of the value as the error's detail message.
    - The purpose of the detail message is to capture and communicate the details of the assertion failure. The message should allow you to diagnose and ultimately fix the error that led the assertion to fail.
    - Note that the ***detail message is not a user-level error message***, so it ***is generally unnecessary to make these messages understandable in isolation, or to internationalize them***.
    - The detail message is meant to be interpreted in the context of a full stack trace, in conjunction with the source code containing the failed assertion.
    - Like all uncaught exceptions, assertion failures are generally labeled in the stack trace with the file and line number from which they were thrown.
    - The second form of the assertion statement should be used in preference to the first only when the program has some additional information that might help diagnose the failure.
- In some cases *Expression*$_1$ may be expensive to evaluate.
- To ensure that assertions are not a performance liability in deployed applications, assertions can be enabled or disabled when the program is started, and are ***disabled by default. Disabling assertions eliminates their performance penalty entirely***.
- Once disabled, they are essentially equivalent to ***empty statements*** in semantics and performance.

Putting Assertions Into Your Code

- situations where it is good to use assertions.
- Internal Invariants
- Control-Flow Invariants
- Preconditions, Postconditions, and Class Invariants

❖ There are also a few situations where you should *not* use them:

- Do *not* use assertions for argument checking in public methods.
    - Argument checking is typically part of the published specifications (or *contract*) of a method, and these specifications must be obeyed whether assertions are enabled or disabled.

- Another problem with using assertions for argument checking is that erroneous arguments should result in an appropriate runtime exception (such as IllegalArgumentException, IndexOutOfBoundsException, or NullPointerException). An assertion failure will not throw an appropriate exception.

- **Do *not* use assertions to do any work that your application requires for correct operation.**
  - Because assertions may be disabled, programs must not assume that the boolean expression contained in an assertion will be evaluated.

```
// Broken! - action is contained in assertion
assert names.remove(null);
// Fixed - action precedes assertion
boolean nullsRemoved = names.remove(null);
assert nullsRemoved;  // Runs whether or not asserts are enabled
```

- As a rule, the ***expressions contained in assertions should be free of side effects***:
  - evaluating the expression should not affect any state that is visible after the evaluation is complete.
  - One exception to this rule is that assertions can modify state that is used only from within other assertions.

Internal Invariants

Use an assertion whenever you would have written a comment that asserts an invariant. For example, you should rewrite the previous if-statement like this:

```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else {
    assert i % 3 == 2 : i;
    ...
}
```

Note, incidentally, that the assertion in the above example may fail if i is negative, as the % operator is not a true *modulus* operator, but computes the *remainder*, which may be negative.

❖ Another good candidate for an assertion is a ***switch statement with no default case***.
- The absence of a default case typically indicates that a programmer believes that one of the cases will always be executed.
- The assumption that a particular variable will have one of a small number of values is an invariant that should be checked with an assertion.

```
default:    assert false : suit;
```

If the suit variable takes on another value and assertions are enabled, the assert will fail and an AssertionError will be thrown.

- An acceptable alternative is:

```
default:    throw new AssertionError(suit);
```

- ***This alternative offers protection even if assertions are disabled, but the extra protection adds no cost:*** the throw statement won't execute unless the program has failed. Moreover, the alternative is legal under some circumstances where the assert statement is not. If the enclosing method returns a value, each case in the switch statement contains a return statement, and no return statement follows the switch

statement, then it would cause a syntax error to add a default case with an assertion. (The method would return without a value if no case matched and assertions were disabled.)

Control-Flow Invariants

- ***place an assertion at any location you assume will not be reached***.
- The assertions statement to use is: assert false;

```
void foo() {
      for (...) {
          if (...)
              return;
      }
      assert false; // Execution should never reach this point!
  }
```

- **Note:** Use this technique with discretion. If a statement is unreachable as defined in the Java Language Specification , you will get a compile time error if you try to assert that it is not reached. Again, an acceptable alternative is simply to throw an AssertionError.

Preconditions, Postconditions, and Class Invariants

- While the assert construct is not a full-blown *design-by-contract* facility, it can help support an informal design-by-contract style of programming. This section shows you how to use asserts for:

  - Preconditions — what must be true when a method is invoked.
    - Lock-Status Preconditions — preconditions concerning whether or not a given lock is held.
  - Postconditions — what must be true after a method completes successfully.
  - Class invariants — what must be true about each instance of a class.

Preconditions

- By convention, preconditions on *public* methods are enforced by explicit checks that throw particular, specified exceptions.
- ***Do not use assertions to check the parameters of a public method.***
  - An assert is inappropriate because the method guarantees that it will always enforce the argument checks.
  - It must check its arguments whether or not assertions are enabled.
  - Further, the assert construct does not throw an exception of the specified type. It can throw only an AssertionError.
- ***You can, however, use an assertion to test a nonpublic method's precondition that you believe will be true no matter what a client does with the class.***

Lock-Status Preconditions

- ***Classes designed for multithreaded use often have non-public methods with preconditions relating to whether or not some lock is held.***
- Note that it is also possible to write a lock-status assertion asserting that a given lock *isn't* held.
- For example, it is not uncommon to see something like this:

```
// Recursive helper method - always called with a lock on this.
private int find(Object key, Object[] arr, int start, int len) {
   assert Thread.holdsLock(this); // lock-status assertion
```

```
        ...
    }
```

Postconditions

- ***You can test postcondition with assertions in both public and nonpublic methods.***

```
public BigInteger modInverse(BigInteger m) {
    if (m.signum <= 0)
        throw new ArithmeticException("Modulus not positive: " + m);
    ... // Do the computation
    assert this.multiply(result).mod(m).equals(ONE) : this;
    return result;
}
```

Occasionally it is necessary to save some data prior to performing a computation in order to check a postcondition. You can do this with two assert statements and a simple inner class that saves the state of one or more variables so they can be checked (or rechecked) after the computation.

Class Invariants

- A class invariants is a type of **internal invariant** that applies to every instance of a class at all times, except when an instance is in transition from one consistent state to another.
- A class invariant can specify the relationships among multiple attributes, and should be true before and after any method completes.
- For example, suppose you implement a balanced tree data structure of some sort. A class invariant might be that the tree is balanced and properly ordered.

The assertion mechanism does not enforce any particular style for checking invariants. It is sometimes convenient, though, to combine the expressions that check required constraints into a single internal method that can be called by assertions. Continuing the balanced tree example, it might be appropriate to implement a private method that checked that the tree was indeed balanced as per the dictates of the data structure:

```
// Returns true if this tree is properly balanced
private boolean balanced() {
    ...
}
```

Because this method checks a constraint that should be true before and after any method completes, each public method and constructor should contain the following line immediately prior to its return:

```
assert balanced();
```

It is generally unnecessary to place similar checks at the head of each public method unless the data structure is implemented by native methods. In this case, it is possible that a memory corruption bug could corrupt a "native peer" data structure in between method invocations. A failure of the assertion at the head of such a method would indicate that such memory corruption had occurred. Similarly, it may be advisable to include class invariant checks at the heads of methods in classes whose state is modifiable by other classes. (Better yet, design classes so that their state is not directly visible to other classes!)

Removing all Trace of Assertions from Class Files

- strip assertions out of class files entirely.
  - this makes it impossible to enable assertions in the field,
  - it also reduces class file size, possibly leading to improved class loading performance.
  - In the absence of a high quality JIT, it could lead to decreased footprint and improved runtime performance.

- *The assertion facility offers no direct support for stripping assertions out of class files.*
- *The assert statement may, however, be used in conjunction with the "conditional compilation" idiom* enabling the compiler to eliminate all traces of these asserts from the class files that it generates:

```
static final boolean asserts = ... ; // false to eliminate asserts

if (asserts)
    assert <expr> ;
```

Requiring that Assertions are Enabled

- Programmers of certain critical systems might wish to ensure that assertions are not disabled in the field.
- The following static initialization idiom prevents a class from being initialized if its assertions have been disabled:

```
static {
    boolean assertsEnabled = false;
    assert assertsEnabled = true; // Intentional side effect!!!
    if (!assertsEnabled)
        throw new RuntimeException("Asserts must be enabled!!!");
}
```

Put this static-initializer at the top of your class.

Compiling Files That Use Assertions

- javac **-source 1.4** MyClass.java
- This flag is necessary so as not to cause source compatibility problems.

Enabling and Disabling Assertions

- By default, assertions are disabled at runtime.
- Two command-line switches allow you to selectively enable or disable assertions.
- To enable assertions at various granularities, use the -enableassertions, or -ea, switch.
- To disable assertions at various granularities, use the -disableassertions, or -da, switch.
- You specify the granularity with the arguments that you provide to the switch:
  - no arguments
    Enables or disables assertions in all classes except system classes.
  - *packageName*...
    Enables or disables assertions in the named package and any subpackages.
  - ...
    Enables or disables assertions in the unnamed package in the current working directory.
  - *className*
    Enables or disables assertions in the named class

- For example, the following command runs a program, BatTutor, with assertions enabled in only package com.wombat.fruitbat and its subpackages:

  java -ea:com.wombat.fruitbat... BatTutor
- ***If a single command line contains multiple instances of these switches, they are processed in order before loading any classes***.
- The above switches apply to all class loaders. With one exception, they also apply to *system classes* (which do not have an explicit class loader). The exception concerns the switches with no arguments, which (as indicated above) do not apply to system classes. This behavior makes it easy to enable asserts in all classes except for system classes, which is commonly desirable.
- To enable assertions in all system classes, use a different switch: -enablesystemassertions, or -esa.
- To disable assertions in system classes, use -disablesystemassertions, or -dsa.
- For example, the following command runs the BatTutor program with assertions enabled in system classes, as well as in the com.wombat.fruitbat package and its subpackages:

```
java -esa -ea:com.wombat.fruitbat...
```

- ***The assertion status of a class (enabled or disabled) is set at the time it is initialized, and does not change.*** There is, however, one corner case that demands special treatment. It is possible, though generally not desirable, to execute methods or constructors prior to initialization. This can happen when a class hierarchy contains a circularity in its static initialization.
- If an assert statement executes before its class is initialized, the execution must behave as if assertions were enabled in the class.

Compatibility With Existing Programs

- The addition of the assert keyword to the Java programming language does not cause any problems with preexisting binaries (.class files).
- If you try to compile an application that uses assert as an identifier, however, you will receive a warning or error message.
- In order to ease the transition from a world where assert is a legal identifier to one where it isn't, the compiler supports two modes of operation in this release:
  - **source mode 1.3** (default) — the compiler accepts programs that use assert as an identifier, but issues warnings. In this mode, programs are *not* permitted to use the assert statement.
  - **source mode 1.4** — the compiler generates an error message if the program uses assert as an identifier. In this mode, programs *are* permitted to use the assert statement.
- *Unless you specifically request source mode 1.4 with the -source 1.4 flag, the compiler operates in source mode 1.3. If you forget to use this this flag, programs that use the new assert statement will not compile.*
- AssertionError a subclass of Error rather than RuntimeException

Switch	Example	Meaning
-ea	Java -ea	Enable assertions by default
-da	Java -da	Disable assertions by default
-ea:<classname>	Java -ea:AssertPackageTest	Enable assertions in class AssertPackageTest
-da:<classname>	Java -da:AssertPackageTest	Disable assertions in class AssertPackageTest
-ea:<packagename>...	- Java –ea:pkg0...	Enable assertions in package pkg0
-da:<packagename>...	Java –da:pkg0...	Disable assertions in package pkg0
-esa	Java –esa	Enable assertions in system classes

-dsa	Java -dsa	Disable assertions in system classes

Assertion do's	Assertion don'ts
Do use to enforce internal assumptions about aspects of data structures	Don't use to enforce command-line usage
Do use to enforce constraints on arguments to private methods	Don't use to enforce constraints on arguments to public methods
Do use to check conditions at the end of any kind of method	Don't use to enforce public usage patterns or protocols
Do use to check for conditional cases that should never happen	Don't use to enforce a property of a piece of user supplied information
Do use to check for conditional cases that should never happen, even if you're really sure they can never happen	Don't use as a shorthand for if (something) error();
Do use to check related conditions at the start of any method	Don't use as an externally controllable conditional
Do use to check things in the middle of a long lived loop	Don't use as a check on the correctness of your compiler, operating system, or hardware, unless you have a specific reason to believe there is something wrong with it and are in the process of debugging it
Do use in lieu of nothing	

Multi-Threading

- ✓ Simultaneously running two or more parts of the same program is called multithreading.
- ✓ Developing a program which can execute multiple tasks simultaneously is called multithreaded programming. Multithreading enables one to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
- ✓ 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.

2 Ways of Creating 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.

2 ways by which threads can be stopped

1. The stop() method: This method stops the thread on which it is invoked. This method is deprecated as it might cause serious system failures.
2. The interrupt() method: This method interrupts the execution flow of the thread on which it is invoked. When the thread interrupts, it may do some other tasks or stop right away.

1. Pre-emptive Scheduling.

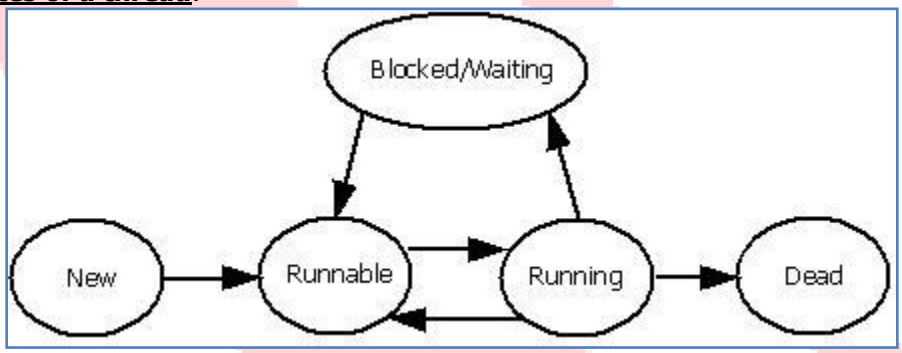
Ways for a thread to leave running state -

- ✓ 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.
- ✓ Macintosh 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 must re-acquire 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)

Transitioning Between Thread States

- ✓ Once a new thread is started, it will always enter the runnable state.
- ✓ The thread scheduler can move a thread back and forth between the runnable state and the running state.
- ✓ For a typical single-processor machine, only one thread can be running at a time, although many threads may be in the runnable state.
- ✓ There is no guarantee that the order in which threads were started determines the order in which they'll run.
- ✓ There's no guarantee that threads will take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation. If you want a guarantee that your threads will take turns regardless of the underlying JVM, you can use the sleep() method. This prevents one thread from hogging the running process while another thread starves. (In most cases, though, yield() works well enough to encourage your threads to play together nicely.)
- ✓ A running thread may enter a blocked/waiting state by a wait(), sleep(), or join() call.
- ✓ A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.
- ✓ When the sleep or wait is over, or an object's lock becomes available, the thread can only reenter the runnable state. It will go directly from waiting to running (well, for all practical purposes anyway).
- ✓ A dead thread cannot be started again.

Sleep, Yield, and Join

- ✓ Sleeping is used to delay execution for a period of time, and no locks are released when a thread goes to sleep.
- ✓ A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the sleep() method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running.
- ✓ The sleep() method is a static method that sleeps the currently executing thread's state. One thread cannot tell another thread to sleep.
- ✓ The setPriority() method is used on Thread objects to give threads a priority of between 1 (low) and 10 (high), although priorities are not guaranteed, and not all JVMs recognize 10 distinct priority levels—some levels may be treated as effectively equal.
- ✓ If not explicitly set, a thread's priority will have the same priority as the priority of the thread that created it.
- ✓ The yield() method may cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out there will be a different thread selected to run. A thread might yield and then immediately reenter the running state.
- ✓ The closest thing to a guarantee is that at any given time, when a thread is running it will usually not have a lower priority than any thread in the runnable state. If a low-priority thread is running when a high-priority thread enters runnable, the JVM will usually preempt the running low-priority thread and put the high-priority thread in.
- ✓ When one thread calls the join() method of another thread, the currently running thread will wait until the thread it joins with has completed. Think of the join() method as saying, "Hey thread, I want to join on to the end of you. Let me know when you're done, so I can enter the runnable state."

Concurrent Access Problems and Synchronized Threads

- ✓ synchronized methods prevent more than one thread from accessing an object's critical method code simultaneously.
- ✓ You can use the synchronized keyword as a method modifier, or to start a synchronized block of code.
- ✓ To synchronize a block of code (in other words, a scope smaller than the whole method), you must specify an argument that is the object whose lock you want to synchronize on.
- ✓ While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's unsynchronized code. When a thread goes to sleep, its locks will be unavailable to other threads.
- ✓ static methods can be synchronized, using the lock from the java.lang.Class instance representing that class.

Communicating with Objects by Waiting and Notifying

- ✓ The wait() method lets a thread say, "there's nothing for me to do now, so put me in your waiting pool and notify me when something happens that I care about." Basically, a wait() call means "wait me in your pool," or "add me to your waiting list."
- ✓ The notify() method is used to send a signal to one and only one of the threads that are waiting in that same object's waiting pool.
- ✓ The notify() method can NOT specify which waiting thread to notify.
- ✓ The method notifyAll() works in the same way as notify(), only it sends the signal to all of the threads waiting on the object.
- ✓ All three methods—wait(), notify(), and notifyAll()—must be called from within a synchronized context! A thread invokes wait() or notify() on a particular object, and the thread must currently hold the lock
- ✓ on that object.

Deadlocked Threads

- ✓ Deadlocking is when thread execution grinds to a halt because the code is waiting for locks to be removed from objects.
- ✓ Deadlocking can occur when a locked object attempts to access another locked object that is trying to access the first locked object. In other words,
- ✓ both threads are waiting for each other's locks to be released; therefore, the locks will never be released!
- ✓ Deadlocking is bad. Don't do it.

Recognize conditions that might prevent a thread from executing.

The following conditions may prevent a thread from executing:

- ✓ The run () method has completed and returned.
- ✓ An unchecked exception has been thrown from within the run () method and it is unhandled.
- ✓ The JVM has exited due to any reason.
- ✓ A deadlock has occurred.
- ✓ The thread has been stopped from executing by the program itself.

Synchronization:

- ✓ When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used only by one thread at a time. The process by which this is ensured is called Synchronization.
- ✓ Key to synchronization is the concept of the monitor (also called Semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. A thread that owns a monitor can reenter the same monitor if it so desires. There are two ways to synchronize your code:

1. Using synchronized methods

- ✓ Synchronizing is easy in Java because all objects have their own monitor associated with them. To enter an object's monitor, just call a method that has been modified with the synchronized keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.
- ✓ When there is no synchronization, nothing exists to stop all the threads from calling the same method on the same object at the same time. This is known as a race condition, because the threads are vying with each other to complete the method. In most situations, a race condition is less predictable, because you cannot be sure when the context switch will occur. This can cause a program to run right one time and wrong the next.
- ✓ To fix this problem you must serialize access to a method. That is, you must restrict its access to only one thread at a time. To do this, qualify the method with 'synchronized' keyword. Anytime that you have a method, or a group of methods that manipulate the internal state of an object in a multi-threaded situation, you should use the 'synchronized' keyword to guard the state from race conditions. Once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, non- synchronized method on that instance will continue to be callable.

2. The synchronized statement

If you want to synchronize access to objects of a class that was not designed for multi-threaded access i.e. the class does not use synchronized methods or if you do not access to the source code of the class, then simply put calls to the methods defined by this class inside a synchronized block

```
synchronized ( object){
        // statements to be synchronized.
 }
```

Here object is a reference to the object being synchronized. If you want to synchronize only a single statement, curly braces are not needed. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Inter-thread communication:

- ✓ Multithreading replaces event loop programming by dividing your tasks into discreet and logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time.
- ✓ To avoid polling, Java includes an elegant inter-process communication via wait(), notify() and notifyAll(). These methods are implemented as final methods in Object, so all classes have them. All these three methods can be called only from within a synchronized method.
- ✓ The method wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the monitor and calls notify(). Additional forms of wait() exist that allow you to specify a period of time to wait.
- ✓ The method notify() wakes up the first thread that called wait() on the same object.
- ✓ The method notifyAll() wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

Deadlock

- ✓ A special type of error in multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object x and another thread enters the monitor on object y. If the thread in x tries to call any synchronized method on y, it will block as expected. However, if the thread in y, in turn, tries to call any synchronized method on x, the thread waits forever, because to access x, it would have to release its own lock on y so that the first thread could complete.
- ✓ Deadlock is a difficult error to debug because:
- ✓ In general, it occurs only rarely, when the two threads time-slice in just the right way.
- ✓ It may involve more than two threads and two synchronized objects.

Main thread:

- ✓ When a Java program starts up, one thread begins running immediately. This thread is the one that is executed when your program begins. The main thread is important for two reasons:
- ✓ It is the thread from which other child threads will be spawned.
- ✓ It must be the last thread to finish execution. When the main thread stops, your program terminates.
- ✓ The main thread is created automatically when a program is started. It can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the method currentThread(), which is a public, static member of Thread.
- ✓ Thread t=Thread.currentThread();
- ✓ When t is used as an argument to println(), it displays the name of the thread, its priority and the name of its group in that order.
- ✓ A thread group is a data structure that controls the state of a collection of threads as a whole. This process is managed by the particular runtime environment.

Return of run ():

- ✓ When the run() returns, the thread has finished its task and is considered dead. There is no way to restart it. If you want the thread's task to be performed again, you have to construct and start a new thread instance. The dead thread continues to exist; it is an object like any other object, and you can still access its data and call its methods. You just cannot make it run again.
- ✓ Instead of using stop() of thread class, if a thread might need to be killed from another thread, then you should send it an interrupt() from the killing method.

Important Exceptions:

- ✓ **IllegalThreadStateException**: thrown to indicate that a thread is not in an appropriate state for the requested operation.
- ✓ **IllegalMonitorStateException**: thrown to indicate that a thread has attempted to wait on an object's monitor without owning the specified monitor.

Why stop (), suspend () and resume () are deprecated?

- ✓ **stop ()** is inherently unsafe. Stopping a thread causes it to unlock all the monitors that it has locked. (the monitors are unlocked as the ThreadDeath exception propagated up the stack.) If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be damaged. When threads operate on damaged objects, arbitrary behavior can result. This behavior may be subtle and difficult to detect, or it may be pronounced. Unlike other unchecked exceptions, ThreadDeath kills threads silently; thus, the user has no warning that his program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even after hours or days in the future. Most uses of stop () should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. To ensure prompt communication of the stop () request, the variable must be volatile (or access to the variable must be synchronized.
- ✓ **suspend ()** is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume (), deadlock results. Such deadlocks typically manifest themselves as frozen processes.
- ✓ **resume ()** exists solely for suspend (). Therefore it has been deprecated too.

Deamon Thread

- ✓ A thread is either a user thread or a deamon thread. t.setDeamon(true); creates a deamon thread
- ✓ setDeamon has to be called before the thread is started
- ✓ The JVM exits if all running threads are deamon threads

Important Facts:

- ✓ A "lock" is a part of any object. One object has only one lock but it may be acquired multiple times (but only by the same thread which already has got it for the first time). If a thread acquires the lock twice then it should release it twice.
- ✓ For static blocks (where there is no instance), there is a class object for that class which has a lock.
- ✓ It's the thread (not a Thread object but the flow of control) that 'acquires' lock. Understand the distinction between a Thread object and a thread. Thread object is just another object. A thread is the flow of control that executes the code. You need a Thread object to create a thread.
- ✓ As there is only one lock for one object, only one thread can get the lock for an object at any given time.

Points to remember:

- ✓ The thread that is calling wait/notify/notifyall on an object MUST have the lock of that object otherwise an IllegalMonitorState exception will be thrown. In other words, acquiring lock of one object and calling notify() on another DOES NOT WORK.
- ✓ When a thread tries to enter a synchronized method/block, it waits till it acquires the lock for the object whose method it is trying to enter. For static methods, it waits for the class object's lock.
- ✓ A thread dies when it's run method ends (or if the stop method, which is deprecated) is called. It cannot be restarted.
- ✓ Methods of a Thread object can be called anytime as if it were just another normal object. Except start() which can be called only once. Calling start() creates a new thread of execution.
- ✓ A thread spawned by a daemon thread is a daemon thread but you can change it by calling setDaemon(false).
- ✓ A thread can be made a daemon thread by calling setDaemon(true) method. This method must be called before the thread is started, otherwise an IllegalThreadStateException will be thrown.
- ✓ Threads have priorities. Thread class defines the int constants MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY. Their values are 10, 0 and 5 but you should use the constant names instead of the values.
- ✓ A newly created thread has the same priority as the thread which created it. You can change it by calling setPriority().
- ✓ Which thread is scheduled when depends on the JVM and platform. So, you can NEVER say for sure about which thread would be running at at what time. Ie. If you start 2 threads you can't say anything about their execution schedule. And your code should not depend on any such assumptions.
- ✓ wait() and sleep() must be enclosed in a try/catch block as they throw InterruptedException.
- ✓ A thread can obtain multiple locks on the same object or multiple objects. If a thread acquires a lock for the same object twice, it should release it twice otherwise the object will remain locked.
- ✓ A thread owning the lock of an object can call other synchronous methods on the same object. In a sense, it is acquiring the same lock more than once.
- ✓ Synchronized methods can be overridden to be non-synchronized. But it does not change the behavior for the super class's method.
- ✓ Beware of deadlock: Consider this situation: Thread1 gets the lock for object1 and tries to acquire the lock for object2. Just before it could get the lock for obj2, the OS preempts this thread and runs another thread t2. Now t2 gets the lock for obj2 (which was available as T1 was stopped just before it could acquire the lock) and then tries to get the lock for Object1 (which was already acquired by T1). Here, you can see that T1 is waiting for obj2's lock which is acquired by T2, and T2 is waiting for obj1's lock which is accuired by T1. Neither of the threads is able to proceed. This is a Deadlock.
- ✓ Java does not provide any mechanism to detect, avoid or solve a deadlock. You must program so that deadlocks don't happen at runtime.

String, StringBuffer & StringBuilder

The String Class

This section covers the String class, and the key concept to understand is that once a String object is created, it can never be changed—so what is happening when a String object seems to be changing? Let's find out.

Strings Are Immutable Objects

We'll start with a little background information about strings. You may not need this for the test, but a little context will help. Handling "strings" of characters is a fundamental aspect of most programming languages. In Java, each character in a string is a 16-bit Unicode character. Because Unicode characters are 16 bits (notthe skimpy 7 or 8 bits that ASCII provides), a rich, international set of characters is easily represented in Unicode.

In Java, strings are objects. Just like other objects, you can create an instance of a String with the new keyword, as follows:

```
String s = new String();
```

This line of code creates a new object of class String, and assigns it to the reference variable s. So far, String objects seem just like other objects. Now, let's give the String a value:

```
s = "abcdef";
```

As you might expect, the String class has about a zillion constructors, so you can use a more efficient shortcut:

```
String s = new String("abcdef");
```

And just because you'll use strings all the time, you can even say this:

```
String s = "abcdef";
```

There are some subtle differences between these options that we'll discuss later, but what they have in common is that they all create a new String object, with a value of "abcdef", and assign it to a reference variable s. Now let's say that you want a second reference to the String object referred to by s:

```
String s2 = s; // refer s2 to the same String as s.
```

So far so good. String objects seem to be behaving just like other objects, so what's all the fuss about?…Immutability! (What the heck is immutability?) Once you have assigned a String a value, that value can never change— it's immutable, frozen solid, won't budge, fini, done. (We'll talk about why later, don't let us forget.)

The good news is that while the String object is immutable, its reference variable is not, so to continue with our previous example:

```
s = s.concat(" more stuff");
// the concat() method 'appends' a literal to the end
```

Now wait just a minute, didn't we just say that Strings were immutable? So what's all this "appending to the end of the string" talk? Excellent question: let's look at what really happened…

The VM took the value of String s (which was "abcdef"), and tacked " more stuff" onto the end, giving us the value "abcdef more stuff". Since Strings are immutable, the VM couldn't stuff this new value into the old String referenced by s, so it created a new String object, gave it the value "abcdef more stuff", and made s refer to it. At this point in our example, we have two String objects: the first one we created, with the value "abcdef", and the second one with the value "abcdef more stuff". Technically there are now three String objects, because the literal argument to concat, " more stuff", is itself a new String object. But we have references only to "abcdef" (referenced by s2) and "abcdef more stuff" (referenced by s).

What if we didn't have the foresight or luck to create a second reference variable for the "abcdef" String before we called s = s.concat(" more stuff");? In that case, the original, unchanged String containing

"abcdef" would still exist in memory, but it would be considered "lost." No code in our program has any way to reference it—it is lost to us. Note, however, that the original "abcdef" String didn't change (it can't, remember, it's immutable); only the reference variable s was changed, so that it would refer to a different String.

To review our first example:

```
String s = "abcdef";
// create a new String object, with value "abcdef", refer s to it
String s2 = s;
// create a 2nd reference variable referring to the same String
// create a new String object, with value "abcdef more stuff",
// refer s to it. (Change s's reference from the old String
// to the new String.) ( Remember s2 is still referring to
// the original "abcdef" String.)
s = s.concat(" more stuff");
```

Important Facts About Strings and Memory

In this section we'll discuss how Java handles String objects in memory, and some of the reasons behind these behaviors.

One of the key goals of any good programming language is to make efficient use of memory. As applications grow, it's very common for String literals to occupy large amounts of a program's memory, and there is often a lot of redundancy within the universe of String literals for a program. To make Java more memory efficient, the JVM sets aside a special area of memory called the "String constant pool." When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created. (The existing String simply has an additional reference.) Now we can start to see why making String objects immutable is such a good idea. If several reference variables refer to the same String without even knowing it, it would be very bad if any of them could change the String's value.

You might say, "Well that's all well and good, but what if someone overrides the String class functionality; couldn't that cause problems in the pool?" That's one of the main reasons that the String class is marked final. Nobody can override the behaviors of any of the String methods, so you can rest assured that the String objects you are counting on to be immutable will, in fact, be immutable.

The StringBuffer and StringBuilder Classes

The java.lang.StringBuffer and java.lang.StringBuilder classes should be used when you have to make a lot of modifications to strings of characters. As we discussed in the previous section, String objects are immutable, so if you choose to do a lot of manipulations with String objects, you will end up with a lot of abandoned String objects in the String pool. (Even in these days of gigabytes of RAM, it's not a good idea to waste precious memory on discarded String pool objects.) On the other hand, objects of type StringBuffer and StringBuilder can be modified over and over again without leaving behind a great effluence of discarded String objects.

StringBuffer vs. StringBuilder

The StringBuilder class was added in Java 5. It has exactly the same API as the StringBuffer class, except StringBuilder is not thread safe. In other words, its methods are not synchronized. (More about thread safety in Chapter 9.) Sun recommends that you use StringBuilder instead of StringBuffer whenever possible because StringBuilder will run faster (and perhaps jump higher). So apart from synchronization, anything we say about StringBuilder's methods holds true for StringBuffer's methods, and vice versa.

Summary:

Strings The most important thing to remember about Strings is that String objects are immutable, but references to Strings are not! You can make a new String by using an existing String as a starting point, but if you don't assign a reference variable to the new String it will be lost to your program—you will have no way to access your new String. Review the important methods in the String class.

The StringBuilder class was added in Java 5. It has exactly the same methods as the old StringBuffer class, except StringBuilder's methods aren't thread-safe. Because StringBuilder's methods are not thread safe, they tend to run faster than StringBuffer methods, so choose StringBuilder whenever threading is not an issue. Both StringBuffer and StringBuilder objects can have their value changed over and over without having to create new objects. If you're doing a lot of string manipulation, these objects will be more efficient than immutable String objects, which are, more or less, "use once, remain in memory forever." Remember, these methods ALWAYS change the invoking object's value, even with no explicit assignment.

Using String, StringBuffer, and StringBuilder

- ✓ String objects are immutable, and String reference variables are not.
- ✓ If you create a new String without assigning it, it will be lost to your program.
- ✓ If you redirect a String reference to a new String, the old String can be lost.
- ✓ String methods use zero-based indexes, except for the second argument of substring().
- ✓ The String class is final—its methods can't be overridden.
- ✓ When the JVM finds a String literal, it is added to the String literal pool.
- ✓ Strings have a method: length(); arrays have an attribute named length.
- ✓ The StringBuffer's API is the same as the new StringBuilder's API, except that StringBuilder's methods are not synchronized for thread safety.
- ✓ StringBuilder methods should run faster than StringBuffer methods.
- ✓ All of the following bullets apply to both StringBuffer and StringBuilder:
- ✓ They are mutable—they can change without creating a new object.
- ✓ StringBuffer methods act on the invoking object, and objects can change without an explicit assignment in the statement.
- ✓ StringBuffer equals() is not overridden; it doesn't compare values.
- ✓ Remember that chained methods are evaluated from left to right.
- ✓ String methods to remember: charAt(), concat(), equalsIgnoreCase(), length(), replace(), substring(), toLowerCase(), toString(), toUpperCase(), and trim().
- ✓ StringBuffer methods to remember: append(), delete(), insert(), reverse(), and toString().

Java.lang Package

The java.lang Package -

- ✓ the java.lang Package contains classes that are fundamental to the Java programming language
- ✓ it is always implicitly imported
- ✓ the most important classes are Object and Class

Object

the Object class is at the root of the class heirarchy, all other classes inherit it's methods

```
protected Object clone() throws CloneNotSupportedException
```
returns an identical copy of an object. The object must implement the Cloneable interface

```
public boolean equals(Object obj)
```
returns true if obj is the same object as the referenced object

```
protected void finalize() throws Throwable
```
called by the garbage collector prior to collecting the object

```
public final Class getClass()
```
returns the runtime class of an object

```
public int hashCode()
```
returns a distinct integer representing a unique object; supports hash tables

```
public final void notify()
```
wakes up a single thread waiting on the object's monitor

```
public final void notifyAll()
```
wakes up all threads waiting on the object's monitor

```
public String toString()
```
returns a string representation of the object

```
public final void wait() throws InterruptedException,
public final void wait(long timeout) throws InterruptedException,
public fianl void wait(long timeout, int nanos) throws InterruptedException
```
causes the current thread to wait until another thread invokes notify() or notifyAll() for this object, or, the specified time elaspses

Class

- ✓ the Class class was introduced in JDK 1.2
- ✓ instances of the Class class represent classes and interfaces in a running Java application
- ✓ also represents arrays, primitive types and void, all of which are Class instances at runtime objects of the Class class are automatically created as classes are loaded by the JVM; they are known as class descriptors
- ✓ provides over 30 methods which can be used to obtain information on a running class. some of the more useful methods are: getName(), toString(), getSuperclass(), isInterface(), newInstance()

Other classes

- ✓ **Wrapper** classes used to represent primitive types as Objects: Boolean, Byte, Short, Character, Integer, Float, Long and Double
- ✓ **Math** class provides commonly used mathematical functions ie cos, sine, tan
- ✓ **String** and **StringBuffer** classes provide commonly used operations on character strings
- ✓ System operation classes: ClassLoader, SecurityManager, Runtime, Process and System which manage the dynamic loading of classes, creation of external processes, security, and host inquiries ie time of day
- ✓ **Package** class is new to JDK 1.2. Provides methods for obtaining package version information stored in the manifest of jar files. Useful methods include: getPackage(), getAllPackages(), which provide package objects that are known to the class loader, and isCompatibleWith() which is used to determine wether a package is comparable to a particular version.
- ✓ all the **Exception** and **Error** classes, including **Throwable**

Interfaces

- ✓ **Cloneable**. Contains no methods. Used to differentiate between objects that are cloneable and non-cloneable.
- ✓ **Comparable**, new in JDK 1.2. Defines the compareTo() method. Objects implementing this interface can be compared and sorted.
- ✓ **Runnable**. Defines the run() method which is invoked when a thread is activated.

- ✓ 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.

Collections & Generics

There are a few basic operations you'll normally use with collections:

- ✓ Add objects to the collection.
- ✓ Remove objects from the collection.
- ✓ Find out if an object (or group of objects) is in the collection.
- ✓ Retrieve an object from the collection (without removing it).
- ✓ Iterate through the collection, looking at each element (object) one after another.

Key Interfaces and Classes of the Collections Framework

For the exam you'll need to know which collection to choose based on a stated requirement. The collections API begins with a group of interfaces, but also gives you a truckload of concrete classes. The core interfaces you need to know for the exam (and life in general) are the following nine:

Collection	Set	SortedSet
List	Map	SortedMap
Queue	NavigableSet	NavigableMap

The core concrete implementation classes you need to know for the exam are the following 13.

Maps	Sets	Lists	Queues	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	LinkedHashSet	Vector		Arrays
TreeMap	TreeSet	LinkedList		

Not all collections in the Collections Framework actually implement the Collection interface. In other words, not all collections pass the IS-A test for Collection. Specifically, none of the Map-related classes and interfaces extend from Collection. So while SortedMap, Hashtable, HashMap, TreeMap, and LinkedHashMap are all thought of as collections, none are actually extended from Collection-with-a-capital-C. To make things a little more confusing, there are really three overloaded uses of the word "collection":

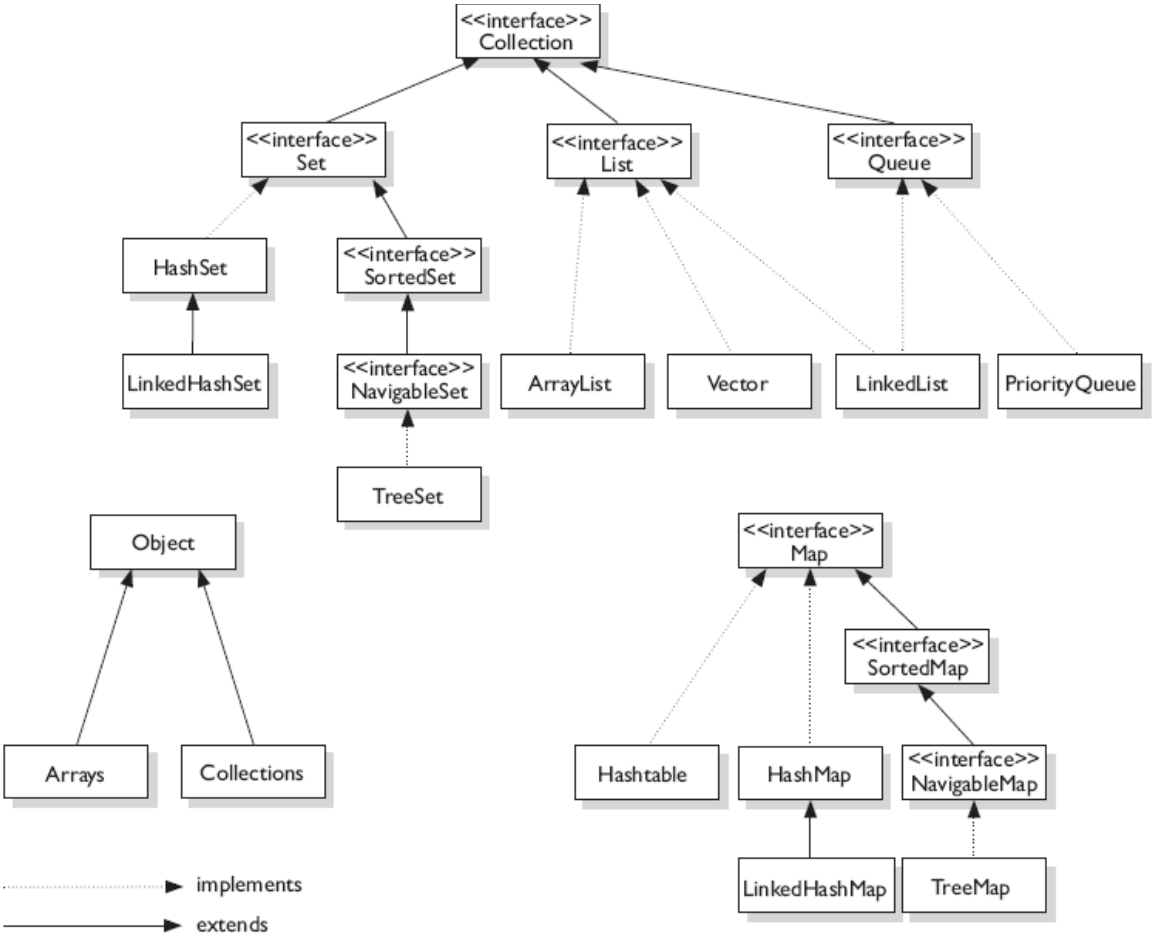
- ✓ collection (lowercase *c*), which represents any of the data structures in which objects are stored and iterated over.
- ✓ Collection (capital *C*), which is actually the java.util.Collection interface from which Set, List, and Queue extend. (That's right, extend, not implement. There are no direct implementations of Collection.)
- ✓ Collections (capital *C* and ends with *s*) is the java.util.Collections class that holds a pile of static utility methods for use with collections.

Collections come in four basic flavors:

- ✓ **Lists** *Lists* of things (classes that implement List).
- ✓ **Sets** *Unique* things (classes that implement Set).
- ✓ **Maps** Things with a *unique* ID (classes that implement Map).
- ✓ **Queues** Things arranged by the order in which they are to be processed.

But there are sub-flavors within those four flavors of collections:

Sorted	Unsorted	Ordered	Unordered



An implementation class can be unsorted and unordered, ordered but unsorted, or both ordered and sorted. But an implementation can never be sorted but unordered, because sorting is a specific type of ordering, as you'll see in a moment. For example, a HashSet is an unordered, unsorted set, while a LinkedHashSet is an ordered (but not sorted) set that maintains the order in which objects were inserted.

Maybe we should be explicit about the difference between sorted and ordered, but first we have to discuss the idea of iteration. When you think of iteration, you may think of iterating over an array using, say, a for loop to access each element in the array in order ([0], [1], [2], and so on). Iterating through a collection usually means walking through the elements one after another starting from the first element.

Sometimes, though, even the concept of first is a little strange—in a Hashtable there really isn't a notion of first, second, third, and so on. In a Hashtable, the elements are placed in a (seemingly) chaotic order based on the hashcode of the key. But something has to go first when you iterate; thus, when you iterate over a Hashtable, there will indeed be an order. But as far as you can tell, it's completely arbitrary and can change in apparently random ways as the collection changes.

Ordered When a collection is ordered, it means you can iterate through the collection in a specific (not-random) order. A Hashtable collection is not ordered. Although the Hashtable itself has internal logic to determine the order (based on hashcodes and the implementation of the collection itself), you won't find any order when you iterate through the Hashtable. An ArrayList, however, keeps the order established by the elements' index position (just like an array). LinkedHashSet keeps the order established by insertion, so the last element inserted is the last element in the LinkedHashSet (as opposed to an ArrayList, where you can insert an element at a specific index position). Finally, there are some collections that keep an order referred to as the natural order of the elements, and those collections are then not just ordered, but also sorted. Let's look at how natural order works for sorted collections.

Sorted A sorted collection means that the order in the collection is determined according to some rule or rules, known as the sort order. A sort order has nothing to do with when an object was added to the collection, or when was the last time it was accessed, or what "position" it was added at. Sorting is done based on properties of the objects themselves. You put objects into the collection, and the collection will figure out what order to put them in, based on the sort order. A collection that keeps an order (such as any List, which uses insertion order) is not really considered sorted unless it sorts using some kind of sort order. Most commonly, the sort order used is something called the natural order. What does that mean?

You know how to sort alphabetically—A comes before B, F comes before G, and so on. For a collection of String objects, then, the natural order is alphabetical. For Integer objects, the natural order is by numeric value—1 before 2, and so on. And for Foo objects, the natural order is…um…we don't know. There is no natural order for Foo unless or until the Foo developer provides one, through an interface (Comparable)that defines how instances of a class can be compared to one another (does instance a come before b, or does instance b come before a?). If the developer decides that Foo objects should be compared using the value of some instance variable (let's say there's one called bar), then a sorted collection will order the Foo objects according to the rules in the Foo class for how to use the bar instance variable to determine the order. Of course, the Foo class might also inherit a natural order from a superclass rather than define its own order, in some cases.

Aside from natural order as specified by the Comparable interface, it's also possible to define other, different sort orders using another interface: Comparator. But for now, just keep in mind that sort order (including natural order) is not the same as ordering by insertion, access, or index. Now that we know about ordering and sorting, we'll look at each of the four interfaces, and we'll dive into the concrete implementations of those interfaces.

List Interface

A List cares about the index. The one thing that List has that non-lists don't have is a set of methods related to the index. Those key methods include things like get(int index), indexOf(Object o), add(int index, Object obj), and so on. All three List implementations are ordered by index position—a position that you determine either by setting an object at a specific index or by adding it without specifying position, in which case the object is added to the end. The three List implementations are described in the following sections.

ArrayList Think of this as a growable array. It gives you fast iteration and fast random access. To state the obvious: it is an ordered collection (by index), but not sorted. You might want to know that as of version 1.4, ArrayList now implements the new RandomAccess interface—a marker interface (meaning it has no methods) that says, "this list supports fast (generally constant time) random access." Choose this over a LinkedList when you need fast iteration but aren't as likely to be doing a lot of insertion and deletion.

Vector Vector is a holdover from the earliest days of Java; Vector and Hashtable were the two original collections, the rest were added with Java 2 versions 1.2 and 1.4. A Vector is basically the same as an ArrayList, but Vector methods are synchronized for thread safety. You'll normally want to use ArrayList instead of Vector because the synchronized methods add a performance hit you might not need. And if you do need thread safety, there are utility methods in class Collections that can help. Vector is the only class other than ArrayList to implement RandomAccess.

LinkedList A LinkedList is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another. This linkage gives you new methods (beyond what you get from the List interface) for adding and removing from the beginning or end, which makes it an easy choice for implementing a stack or queue. Keep in mind that a LinkedList may iterate more slowly than an ArrayList, but it's a good choice when you need fast insertion and deletion. As of Java 5, the LinkedList class has been enhanced to implement the java.util.Queue interface. As such, it now supports the common queue methods: peek(), poll(), and offer().

Set Interface
A Set cares about uniqueness—it doesn't allow duplicates. Your good friend the equals() method determines whether two objects are identical (in which case only one can be in the set). The three Set implementations are described in the following sections.

HashSet A HashSet is an unsorted, unordered Set. It uses the hashcode of the object being inserted, so the more efficient your hashCode() implementation the better access performance you'll get. Use this class when you want a collection with no duplicates and you don't care about order when you iterate through it.

When using HashSet or LinkedHashSet, the objects you add to them must override hashCode(). If they don't override hashCode(), the default Object. hashCode() method will allow multiple objects that you might consider "meaningfully equal" to be added to your "no duplicates allowed" set.

LinkedHashSet A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements. Use this class instead of HashSet when you care about the iteration order. When you iterate through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.

TreeSet The TreeSet is one of two sorted collections (the other being TreeMap). It uses a Red-Black tree structure (but you knew that), and guarantees that the elements will be in ascending order, according to natural order. Optionally, you can construct a TreeSet with a constructor that lets you give the collection your own rules for what the order should be (rather than relying on the ordering defined by the elements' class) by using a Comparable or Comparator. As of Java 6, TreeSet implements NavigableSet.

Map Interface
A Map cares about unique identifiers. You map a unique key (the ID) to a specific value, where both the key and the value are, of course, objects. You're probably quite familiar with Maps since many languages support data structures that use a key/value or name/value pair. The Map implementations let you do things like search for a value based on the key, ask for a collection of just the values, or ask for a collection of just the keys. Like Sets, Maps rely on the equals() method to determine whether two keys are the same or different.

HashMap The HashMap gives you an unsorted, unordered Map. When you need a Map and you don't care about the order (when you iterate through it), then HashMap is the way to go; the other maps add a little more overhead. Where the keys land in the Map is based on the key's hashcode, so, like HashSet, the more efficient your hashCode() implementation, the better access performance you'll get. HashMap allows one null key and multiple null values in a collection.

Hashtable Like Vector, Hashtable has existed from prehistoric Java times. For fun, don't forget to note the naming inconsistency: HashMap vs. Hashtable. Where's the capitalization of t? Oh well, you won't be expected to spell it. Anyway, just as Vector is a synchronized counterpart to the sleeker, more modern ArrayList, Hashtable is the synchronized counterpart to HashMap. Remember that you don't synchronize a class, so when we say that Vector and Hashtable are synchronized, we just mean that the key methods of the class are synchronized. Another difference, though, is that while HashMap lets you have null values as well as one null key, a Hashtable doesn't let you have anything that's null.

LinkedHashMap Like its Set counterpart, LinkedHashSet, the LinkedHash- Map collection maintains insertion order (or, optionally, access order). Although it will be somewhat slower than HashMap for adding and removing elements, you can expect faster iteration with a LinkedHashMap.

TreeMap You can probably guess by now that a TreeMap is a sorted Map. And you already know that by default, this means "sorted by the natural order of the elements." Like TreeSet, TreeMap lets you define a custom sort order (via a Comparable or Comparator) when you construct a TreeMap, that specifies how the elements should be compared to one another when they're being ordered. As of Java 6, TreeMap implements NavigableMap.

The following Table summarizes the 11 of the 13 concrete collection-oriented classes you'll need to understand for the exam.

Class	Map	Set	List	Ordered	Sorted
HashMap	X			No	No
Hashtable	x			No	No
TreeMap	x			Sorted	By *natural order* or custom comparison rules
LinkedHashMap	x			By insertion order or last access order	No
HashSet		x		No	No
TreeSet		x		Sorted	By *natural order* or custom comparison rules
LinkedHashSet		x		By insertion order	No
ArrayList			x	By index	No
Vector			x	By index	No
LinkedList			x	By index	No
PriorityQueue				Sorted	By to-do order

ArrayList Basics

The java.util.ArrayList class is one of the most commonly used of all the classes in the Collections Framework. It's like an array on vitamins. Some of the advantages ArrayList has over arrays are

- ✓ It can grow dynamically.
- ✓ It provides more powerful insertion and search mechanisms than arrays.

Autoboxing with Collections

In general, collections can hold Objects but not primitives. Prior to Java 5, a very common use for the wrapper classes was to provide a way to get a primitive into a collection. Prior to Java 5, you had to wrap a primitive by hand before you could put it into a collection. With Java 5, primitives still have to be wrapped, but autoboxing takes care of it for you.

```
List myInts = new ArrayList(); // pre Java 5 declaration
myInts.add(new Integer(42)); // had to wrap an int
As of Java 5 we can say
myInts.add(42); // autoboxing handles it!
```

Sorting Collections and Arrays

Sorting and searching topics have been added to the exam for Java 5. Both collections and arrays can be sorted and searched using methods in the API.

The Comparable Interface

The Comparable interface is used by the Collections.sort() method and the java.util.Arrays.sort() method to sort Lists and arrays of objects, respectively. To implement Comparable, a class must implement a single method, compareTo().

The compareTo() method returns an int with the following characteristics:

- ✓ negative If thisObject < anotherObject
- ✓ zero If thisObject == anotherObject
- ✓ positive If thisObject > anotherObject

Key Interface Methods	List	Set	Map	Description
boolean **add**(element) boolean **add**(index, element)	X X	X		Add an element. For Lists, optionally add the element at an index point.
boolean **contains**(object) boolean **containsKey**(object key) boolean **containsValue**(object value)	X	X	X X	Search a collection for an object (or, optionally for Maps a key), return the result as a boolean.
object **get**(index) object **get**(key)	X		 X	Get an object from a collection, via an index or a key.
int **indexOf**(object)	X			Get the location of an object in a List.
Iterator **iterator**()	X	X		Get an Iterator for a List or a Set.
Set **keySet**()			X	Return a Set containing a Map's keys.
put(key, value)			X	Add a key/value pair to a Map.
remove(index) **remove**(object) **remove**(key)	X X	X	 X	Remove an element via an index, or via the element's value, or via a key.
int **size()**	X	X	X	Return the number of elements in a collection.
Object[] **toArray**() T[] **toArray**(T[])	X	X		Return an array containing the elements of the collection.

Collections

- ✓ Common collection activities include adding objects, removing objects, verifying object inclusion, retrieving objects, and iterating.
- ✓ Three meanings for "collection":
  - collection Represents the data structure in which objects are stored
  - Collection java.util interface from which Set and List extend
  - Collections A class that holds static collection utility methods
- ✓ Four basic flavors of collections include Lists, Sets, Maps, Queues:
  - Lists of things Ordered, duplicates allowed, with an index.
  - Sets of things May or may not be ordered and/or sorted; duplicates not allowed.
  - Maps of things with keys May or may not be ordered and/or sorted; duplicate keys are not allowed.
  - Queues of things to process Ordered by FIFO or by priority.
- ✓ Four basic sub-flavors of collections Sorted, Unsorted, Ordered, Unordered.
  - Ordered Iterating through a collection in a specific, non-random order.
  - Sorted Iterating through a collection in a sorted order.
- ✓ Sorting can be alphabetic, numeric, or programmer-defined.

Key Attributes of Common Collection Classes

- ✓ ArrayList: Fast iteration and fast random access.
- ✓ Vector: It's like a slower ArrayList, but it has synchronized methods.
- ✓ LinkedList: Good for adding elements to the ends, i.e., stacks and queues.
- ✓ HashSet: Fast access, assures no duplicates, provides no ordering.
- ✓ LinkedHashSet: No duplicates; iterates by insertion order.

- ✓ TreeSet: No duplicates; iterates in sorted order.
- ✓ HashMap: Fastest updates (key/values); allows one null key, many null values.
- ✓ Hashtable: Like a slower HashMap (as with Vector, due to its synchronized methods). No null values or null keys allowed.
- ✓ LinkedHashMap: Faster iterations; iterates by insertion order or last accessed; allows one null key, many null values.
- ✓ TreeMap: A sorted map.
- ✓ PriorityQueue: A to-do list ordered by the elements' priority.

Using Collection Classes

- ✓ Collections hold only Objects, but primitives can be autoboxed.
- ✓ Iterate with the enhanced for, or with an Iterator via hasNext() & next().
- ✓ hasNext() determines if more elements exist; the Iterator does NOT move.
- ✓ next() returns the next element AND moves the Iterator forward.
- ✓ To work correctly, a Map's keys must override equals() and hashCode().
- ✓ Queues use offer() to add an element, poll() to remove the head of the queue, and peek() to look at the head of a queue.
- ✓ As of Java 6 TreeSets and TreeMaps have new navigation methods like floor() and higher().
- ✓ You can create/extend "backed" sub-copies of TreeSets and TreeMaps.

Sorting and Searching Arrays and Lists

- ✓ Sorting can be in natural order, or via a Comparable or many Comparators.
- ✓ Implement Comparable using compareTo(); provides only one sort order.
- ✓ Create many Comparators to sort a class many ways; implement compare().
- ✓ To be sorted and searched, a List's elements must be comparable.
- ✓ To be searched, an array or List must first be sorted.

Utility Classes: Collections and Arrays

- ✓ Both of these java.util classes provide
  - A sort() method. Sort using a Comparator or sort using natural order.
  - A binarySearch() method. Search a pre-sorted array or List.
- ✓ Arrays.asList() creates a List from an array and links them together.
- ✓ Collections.reverse() reverses the order of elements in a List.
- ✓ Collections.reverseOrder() returns a Comparator that sorts in reverse.
- ✓ Lists and Sets have a toArray() method to create arrays.

Overriding hashCode() and equals()

- ✓ equals(), hashCode() and toString() are public.
- ✓ Override toString() so that System.out.println() or other methods can see something useful, like your object's state.
- ✓ Use == to determine if two reference variables refer to the same object.
- ✓ Use equals() to determine if two objects are meaningfully equivalent.
- ✓ If you don't override equals(), your objects won't be useful hashing keys.
- ✓ If you don't override equals(), different objects can't be considered equal.
- ✓ Strings and wrappers override equals() and make good hashing keys.
- ✓ When overriding equals(), use the instanceof operator to be sure you're evaluating an appropriate class.
- ✓ When overriding equals(), compare the objects' significant attributes.

- ✓ Highlights of the equals() contract:
  - Reflexive: x.equals(x) is true.
  - Symmetric: If x.equals(y) is true, then y.equals(x) must be true.
  - Transitive: If x.equals(y) is true, and y.equals(z) is true, then z.equals(x) is true.
  - Consistent: Multiple calls to x.equals(y) will return the same result.
  - Null: If x is not null, then x.equals(null) is false.
- ✓ If x.equals(y) is true, then x.hashCode() == y.hashCode() is true.

- ✓ If you override equals(), override hashCode().
- ✓ HashMap, HashSet, Hashtable, LinkedHashMap, & LinkedHashSet use hashing.
- ✓ An appropriate hashCode() override sticks to the hashCode() contract.
- ✓ An efficient hashCode() override distributes keys evenly across its buckets.
- ✓ An overridden equals() must be at least as precise as its hashCode() mate.
- ✓ To reiterate: if two objects are equal, their hashcodes must be equal.
- ✓ It's legal for a hashCode() method to return the same value for all instances (although in practice it's very inefficient).

- ✓ Highlights of the hashCode() contract:
  - Consistent: multiple calls to x.hashCode() return the same integer.
  - If x.equals(y) is true, x.hashCode() == y.hashCode() is true.
  - If x.equals(y) is false, then x.hashCode() == y.hashCode() can be either true or false, but false will tend to create better efficiency.
- ✓ transient variables aren't appropriate for equals() and hashCode().

Generics

- ✓ Generics let you enforce compile-time type safety on Collections (or other classes and methods declared using generic type parameters).
- ✓ An ArrayList<Animal> can accept references of type Dog, Cat, or any other subtype of Animal (subclass, or if Animal is an interface, implementation).
- ✓ When using generic collections, a cast is not needed to get (declared type) elements out of the collection. With non-generic collections, a cast is required:

```
List<String> gList = new ArrayList<String>();
List list = new ArrayList();
// more code
String s = gList.get(0); // no cast needed
String s = (String)list.get(0); // cast required
```

- ✓ You can pass a generic collection into a method that takes a non-generic collection, but the results may be disastrous. The compiler can't stop the method from inserting the wrong type into the previously type safe collection.
- ✓ If the compiler can recognize that non-type-safe code is potentially endangering something you originally declared as type-safe, you will get a compiler warning. For instance, if you pass a List<String> into a method declared as

```
void foo(List aList) { aList.add(anInteger); }
```

  You'll get a warning because add() is potentially "unsafe".
- ✓ "Compiles without error" is not the same as "compiles without warnings." A compilation warning is not considered a compilation error or failure.
- ✓ Generic type information does not exist at runtime—it is for compile-time safety only. Mixing generics with legacy code can create compiled code that may throw an exception at runtime.
- ✓ Polymorphic assignments applies only to the base type, not the generic type parameter. You can say

```
List<Animal> aList = new ArrayList<Animal>(); // yes
```

  You can't say

```
List<Animal> aList = new ArrayList<Dog>(); // no
```

- ✓ The polymorphic assignment rule applies everywhere an assignment can be made. The following are NOT allowed:

```
void foo(List<Animal> aList) { } // cannot take a List<Dog>
List<Animal> bar() { } // cannot return a List<Dog>
```

- ✓ Wildcard syntax allows a generic method, accept subtypes (or supertypes) of the declared type of the method argument:

```
void addD(List<Dog> d) {} // can take only <Dog>
void addD(List<extends Dog>) {} // take a <Dog> or <Beagle>
```

- ✓ The wildcard keyword extends is used to mean either "extends" or "implements." So in <extends Dog>, Dog can be a class or an interface.
- ✓ When using a wildcard, List<extends Dog>, the collection can be accessed but not modified.

- ✓ When using a wildcard, List<?>, any generic type can be assigned to the reference, but for access only, no modifications.
- ✓ List<Object> refers only to a List<Object>, while List<?> or List<extends Object> can hold any type of object, but for access only.
- ✓ Declaration conventions for generics use T for type and E for element:

```
public interface List<E> // API declaration for List
boolean add(E o) // List.add() declaration
```

- ✓ The generics type identifier can be used in class, method, and variable declarations:

```
class Foo<t> { } // a class
T anInstance; // an instance variable
Foo(T aRef) {} // a constructor argument
void bar(T aRef) {} // a method argument
T baz() {} // a return type
```

- ✓ The compiler will substitute the actual type.
- ✓ You can use more than one parameterized type in a declaration:

```
public class UseTwo<T, X> { }
```

- ✓ You can declare a generic method using a type not defined in the class:

```
public <T> void makeList(T t) { }
```

is NOT using T as the return type. This method has a void return type, but to use T within the method's argument you must declare the <T>, which happens before the return type.

Auto-boxing and auto-unboxing?

- Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.
- Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.

Advantages of autoboxing & auto-unboxing.

1. Removes the tedium of manually boxing and unboxing values.
2. Helps prevent errors.
3. Makes working with the Collections Framework much easier.

AWT

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_HORIZONTA	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	Text event

	L_ONLY TextArea.SCROLLBARS_VERTICAL_ONLY	TextArea(String text, int nrows, int ncols, int scroll)	

- 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.

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

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

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.

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.

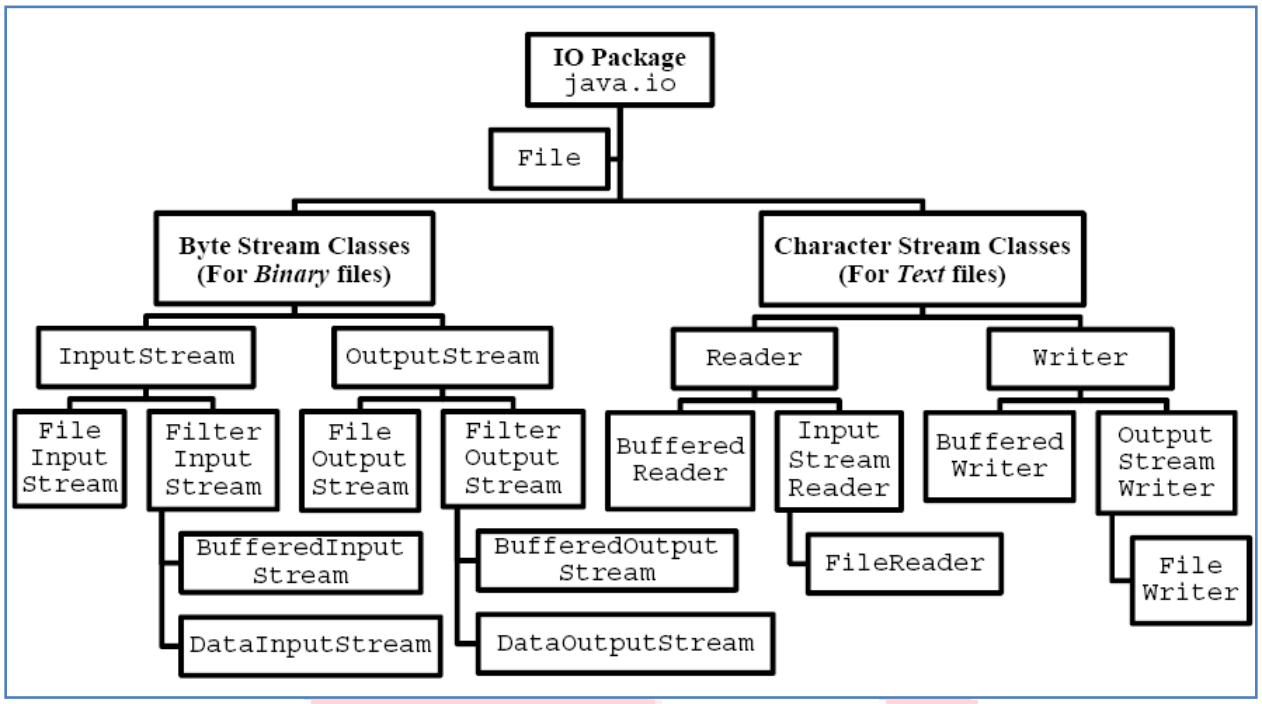
Method	Description
void init()	This method is called only once by the applet context to inform the applet that it has been loaded into the system. Always followed by calls to start() and paint() methods. Same purpose as a constructor. Use this method to perform any initialization.
void start()	Applet 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”.

I/O



Here's a summary of the I/O classes you'll need to understand:

- **File** The API says that the class File is "An abstract representation of file and directory pathnames." The File class isn't used to actually read or write data; it's used to work at a higher level, making new empty files, searching for files, deleting files, making directories, and working with paths.
- **FileReader** This class is used to read character files. Its read() methods are fairly low-level, allowing you to read single characters, the whole stream of characters, or a fixed number of characters. FileReaders are usually wrapped by higher-level objects such as BufferedReaders, which improve performance and provide more convenient ways to work with the data.
- **BufferedReader** This class is used to make lower-level Reader classes like FileReader more efficient and easier to use. Compared to FileReaders, BufferedReaders read relatively large chunks of data from a file at once, and keep this data in a buffer. When you ask for the next character or line of data, it is retrieved from the buffer, which minimizes the number of times that time-intensive, file read operations are performed. In addition,BufferedReader provides more convenient methods such as readLine(), that allow you to get the next line of characters from a file.
- **FileWriter** This class is used to write to character files. Its write() methods allow you to write character(s) or Strings to a file. FileWriters are usually wrapped by higher-level Writer objects such as BufferedWriters or PrintWriters, which provide better performance and higher-level, more flexible methods to write data.
- **BufferedWriter** This class is used to make lower-level classes like FileWriters more efficient and easier to use. Compared to FileWriters, BufferedWriters write relatively large chunks of data to a file at once, minimizing the number of times that slow, file writing operations are performed. The BufferedWriter class also provides a newLine() method to create platform-specific line separators automatically.
- **PrintWriter** This class has been enhanced significantly in Java 5. Because of newly created methods and constructors (like building a PrintWriter with a File or a String), you might find that you can use PrintWriter in places where you previously needed a Writer to be wrapped with a FileWriter and/or a BufferedWriter. New methods like format(), printf(), and append() make PrintWriters very flexible and powerful.

- **Console** This new, Java 6 convenience class provides methods to read input from the console and write formatted output to the console.

Stream classes are used to read and write bytes, and Readers and Writers are used to read and write characters. Since all of the fi le I/O on the exam is related to characters, if you see API class names containing the word "Stream", for instance DataOutputStream, then the question is probably about serialization, or something unrelated to the actual I/O objective.

File I/O

- ✓ The classes you need to understand in java.io are File, FileReader, BufferedReader, FileWriter, BufferedWriter, PrintWriter, and Console.
- ✓ A new File object doesn't mean there's a new file on your hard drive.
- ✓ File objects can represent either a file or a directory.
- ✓ The File class lets you manage (add, rename, and delete) files and directories.
- ✓ The methods createNewFile() and mkdir() add entries to your file system.
- ✓ FileWriter and FileReader are low-level I/O classes. You can use them to write and read files, but they should usually be wrapped.
- ✓ Classes in java.io are designed to be "chained" or "wrapped."
- ✓ It's very common to "wrap" a BufferedReader around a FileReader or a BufferedWriter around a FileWriter, to get access to higher-level (more convenient) methods.
- ✓ PrintWriters can be used to wrap other Writers, but as of Java 5 they can be built directly from Files or Strings.
- ✓ Java 5 PrintWriters have new append(), format(), and printf() methods.
- ✓ Console objects can read non-echoed input and are instantiated using System.console().

Serialization

- ✓ The classes you need to understand are all in the java.io package; they include: ObjectOutputStream and ObjectInputStream primarily, and FileOutputStream and FileInputStream because you will use them to create the low-level streams that the ObjectXxxStream classes will use.
- ✓ A class must implement Serializable before its objects can be serialized.
- ✓ The ObjectOutputStream.writeObject() method serializes objects, and the ObjectInputStream.readObject() method deserializes objects.
- ✓ If you mark an instance variable transient, it will not be serialized even thought the rest of the object's state will be.
- ✓ You can supplement a class's automatic serialization process by implementing the writeObject() and readObject() methods. If you do this, embedding calls to defaultWriteObject() and defaultReadObject(), respectively, will handle the part of serialization that happens normally.
- ✓ If a superclass implements Serializable, then its subclasses do automatically.
- ✓ If a superclass doesn't implement Serializable, then when a subclass object is deserialized, the superclass constructor will be invoked, along with its superconstructor(s).
- ✓ DataInputStream and DataOutputStream aren't actually on the exam, in spite of what the Sun objectives say.

Dates, Numbers, and Currency

- ✓ The classes you need to understand are java.util.Date, java.util.Calendar, java.text.DateFormat, java.text.NumberFormat, and java.util.Locale.
- ✓ Most of the Date class's methods have been deprecated.
- ✓ A Date is stored as a long, the number of milliseconds since January 1, 1970.
- ✓ Date objects are go-betweens the Calendar and Locale classes.
- ✓ The Calendar provides a powerful set of methods to manipulate dates, performing tasks such as getting days of the week, or adding some number of months or years (or other increments) to a date.
- ✓ Create Calendar instances using static factory methods (getInstance()).
- ✓ The Calendar methods you should understand are add(), which allows you to add or subtract various pieces (minutes, days, years, and so on) of dates, and roll(), which works like add() but

doesn't increment a date's bigger pieces. (For example: adding 10 months to an October date changes the month to August, but doesn't increment the Calendar's year value.)

- ✓ DateFormat instances are created using static factory methods (getInstance() and getDateInstance()).
- ✓ There are several format "styles" available in the DateFormat class.
- ✓ DateFormat styles can be applied against various Locales to create a wide array of outputs for any given date.
- ✓ The DateFormat.format() method is used to create Strings containing properly formatted dates.
- ✓ The Locale class is used in conjunction with DateFormat and NumberFormat.
- ✓ Both DateFormat and NumberFormat objects can be constructed with a specific, immutable Locale.
- ✓ For the exam you should understand creating Locales using language, or a combination of language and country.

Parsing, Tokenizing, and Formatting

- ✓ regex is short for regular expressions, which are the patterns used to search for data within large data sources.
- ✓ regex is a sub-language that exists in Java and other languages (such as Perl).
- ✓ regex lets you to create search patterns using literal characters or metacharacters. Metacharacters allow you to search for slightly more abstract data like "digits" or "whitespace".
- ✓ Study the \d, \s, \w, and . metacharacters
- ✓ regex provides for quantifiers which allow you to specify concepts like: "look for one or more digits in a row."
- ✓ Study the ?, *, and + greedy quantifiers.
- ✓ Remember that metacharacters and Strings don't mix well unless you remember to "escape" them properly. For instance String s = "\\d";
- ✓ The Pattern and Matcher classes have Java's most powerful regex capabilities.
- ✓ You should understand the Pattern compile() method and the Matcher matches(), pattern(), find(), start(), and group() methods.
- ✓ You WON'T need to understand Matcher's replacement-oriented methods.
- ✓ You can use java.util.Scanner to do simple regex searches, but it is primarily intended for tokenizing.
- ✓ Tokenizing is the process of splitting delimited data into small pieces.
- ✓ In tokenizing, the data you want is called tokens, and the strings that separate the tokens are called delimiters.
- ✓ Tokenizing can be done with the Scanner class, or with String.split().
- ✓ Delimiters are single characters like commas, or complex regex expressions.
- ✓ The Scanner class allows you to tokenize data from within a loop, which allows you to stop whenever you want to.
- ✓ The Scanner class allows you to tokenize Strings or streams or files.
- ✓ The String.split() method tokenizes the entire source data all at once, so large amounts of data can be quite slow to process.
- ✓ New to Java 5 are two methods used to format data for output. These methods are format() and printf(). These methods are found in the PrintStream class, an instance of which is the out in System.out.
- ✓ The format() and printf() methods have identical functionality.
- ✓ Formatting data with printf() (or format()) is accomplished using formatting strings that are associated with primitive or string arguments.
- ✓ The format() method allows you to mix literals in with your format strings.
- ✓ The format string values you should know are
- ✓ Flags: -, +, 0, "," , and (
- ✓ Conversions: b, c, d, f, and s
- ✓ If your conversion character doesn't match your argument type, an exception will be thrown.

JDBC

JDBC provides a standard library for accessing relational databases

– API standardizes

- Way to establish connection to database
- Approach to initiating queries
- Method to create stored (parameterized) queries
- The data structure of query result (table)
  - – Determining the number of columns
  - – Looking up metadata, etc.
  - – API does not standardize SQL syntax
- JDBC is not embedded SQL

– JDBC classes are in the java.sql package

Note: JDBC is not officially an acronym; unofficially, "Java DataBase Connectivity" is commonly used

JDBC consists of two parts:

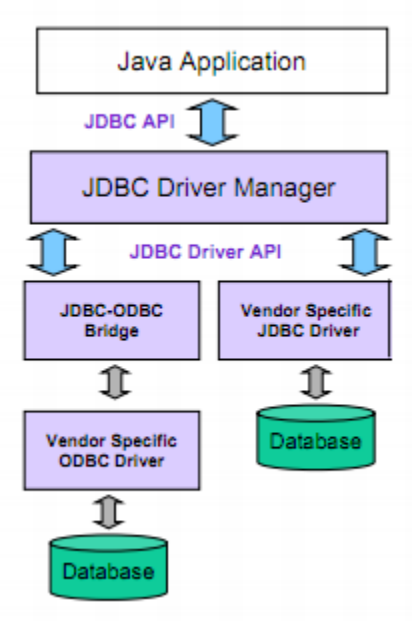
– JDBC API, a purely Java-based API

– JDBC Driver Manager,which communicates with vendor-specific drivers that perform the real communication with the database.

• Point: translation to vendor format is performed on the client

– No changes needed to server

– Driver (translator) needed on client



JDBC Data Types:

JDBC Type	Java Type
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT DOUBLE	double
BINARY VARBINARY LONGVARBINARY	byte[]
CHAR VARCHAR LONGVARCHAR	String

JDBC Type	Java Type
NUMERIC DECIMAL	BigDecimal
DATE	java.sql.Date
TIME TIMESTAMP	java.sql.Timestamp
CLOB	Clob*
BLOB	Blob*
ARRAY	Array*
DISTINCT	mapping of underlying type
STRUCT	Struct*
REF	Ref*
JAVA_OBJECT	underlying Java class

*SQL3 data type supported in JDBC 2.0

Seven Basic Steps in Using JDBC

1. Load the driver
2. Define the Connection URL
3. Establish the Connection
4. Create a Statement object
5. Execute a query
6. Process the results
7. Close the connection

1. Load the driver

```
try {
Class.forName("connect.microsoft.MicrosoftDriver");
Class.forName("oracle.jdbc.driver.OracleDriver");
} catch { ClassNotFoundException cnfe) {
System.out.println("Error loading driver: " cnfe);
}
```

2. Define the Connection URL

```
String host = "dbhost.yourcompany.com";
String dbName = "someName";
int port = 1234;
String oracleURL = "jdbc:oracle:thin:@" + host + ":" + port + ":" + dbName;
String sybaseURL = "jdbc:sybase:Tds:" + host + ":" + port + ":" +
"?SERVICENAME=" + dbName;
```

3. Establish the Connection

```
String username = "jay_debesee";
String password = "secret";
Connection connection = DriverManager.getConnection(oracleURL,
username,password);
```

• Optionally, look up information about the database

```
DatabaseMetaData dbMetaData = connection.getMetaData();
String productName = dbMetaData.getDatabaseProductName();
System.out.println("Database: " + productName);
String productVersion = dbMetaData.getDatabaseProductVersion();
System.out.println("Version: " + productVersion);
```

4. Create a Statement

```
Statement statement = connection.createStatement();
```

5. Execute a Query

```
String query = "SELECT col1, col2, col3 FROM sometable";
ResultSet resultSet = statement.executeQuery(query);
```

– To modify the database, use executeUpdate, supplying a string that uses UPDATE, INSERT, or DELETE
– Use setQueryTimeout to specify a maximum delay to wait for results

6. Process the Result

```
while(resultSet.next()) {
System.out.println(resultSet.getString(1) + " " + resultSet.getString(2) + " " +
resultSet.getString(3));
}
```

– First column has index 1, not 0
– ResultSet provides various getXxx methods that take a column index or column name and returns the data
– You can also access result meta data (column names, etc.)

7. Close the Connection

```
connection.close();
```

– Since opening a connection is expensive, postpone this step if additional database operations are expected

Using Statement

– Through the Statement object, SQL statements are sent to the database.
– Three types of statement objects are available:
- Statement
  – For executing a simple SQL statement
- PreparedStatement
  – For executing a precompiled SQL statement passing in parameters
- CallableStatement
  – For executing a database stored procedure

Useful Statement Methods

- executeQuery
  – Executes the SQL query and returns the data in a table (ResultSet)
  – The resulting table may be empty but never null

```
ResultSet results = statement.executeQuery("SELECT a, b FROM table");
```

- executeUpdate
  – Used to execute for INSERT, UPDATE, or DELETE SQL statements
  – The return is the number of rows that were affected in the database
  – Supports Data Definition Language (DDL) statements
    CREATE TABLE, DROP TABLE and ALTER TABLE

```
int rows = statement.executeUpdate("DELETE FROM EMPLOYEES" + "WHERE STATUS=0");
```

- execute
  – Generic method for executing stored procedures and prepared statements
  – Rarely used (for multiple return result sets)
  – The statement execution may or may not return a ResultSet (use statement.getResultSet). If the return value is true, two or more result sets were produced
- getMaxRows/setMaxRows
  – Determines the maximum number of rows a ResultSet may contain
  – Unless explicitly set, the number of rows is unlimited (return value of 0)
- getQueryTimeout/setQueryTimeout
  – Specifies the amount of a time a driver will wait for a STATEMENT to complete before throwing a SQLException

Prepared Statements (Precompiled Queries)

– If you are going to execute similar SQL statements multiple times, using "prepared" (parameterized) statements can be more efficient
– Create a statement in standard form that is sent to the database for compilation before actually being used
– Each time you use it, you simply replace some of the marked parameters using the setXxx methods
- As PreparedStatement inherits from Statement the corresponding execute methods have no parameters
  – execute()
  – executeQuery()
  – executeUpdate()

Useful Prepared Statement Methods

- setXxx
  – Sets the indicated parameter (?) in the SQL statement to the value
- clearParameters
  – Clears all set parameter values in the statement
- Handling Servlet Data
  – Query data obtained from a user through an HTML form may have SQL or special characters that may require escape sequences
  – To handle the special characters, pass the string to the PreparedStatement setString method which will automatically escape the string as necessary

New features added to JDBC 2.0: The JDBC 2.0 API includes the complete JDBC API, which includes both core and Optional Package API, and provides inductrial-strength database computing capabilities.

- ✓ Scrollable result sets- using new methods in the ResultSet interface allows programmatically move the to particular row or to a position relative to its current position
- ✓ Batch Updates functionality to the java applications.
- ✓ Java applications can now use the ResultSet.updateXXX methods.
- ✓ New data types - interfaces mapping the SQL3 data types
- ✓ Custom mapping of user-defined types (UTDs)
- ✓ Miscellaneous features, including performance hints, the use of character streams, full precision for java.math.BigDecimal values, additional security, and support for time zones in date, time, and timestamp values.

New features added to JDBC 3.0

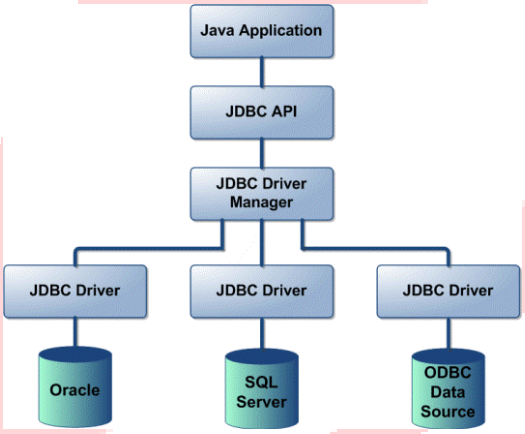
- ✓ Savepoint support
- ✓ Reuse of prepared statements by connection pools
- ✓ Retrieval of auto-generated keys
- ✓ Ability to have multiple open ResultSet objects
- ✓ Ability to make internal updates to the data in Blob and Clob objects
- ✓ Ability to Update columns containing BLOB, CLOB, ARRAY and REF types
- ✓ Both java.sql and javax.sql (JDBC 2.0 Optional Package) are included with J2SE 1.4

New features added to JDBC 4.0

The major features added in JDBC 4.0 include :

- ✓ Auto-loading of JDBC driver class
- ✓ Connection management enhancements
- ✓ Support for RowId SQL type
- ✓ DataSet implementation of SQL using Annotations
- ✓ SQL exception handling enhancements
- ✓ SQL XML support

JDBC Architecture



Types of Drivers

There are four types of drivers defined by JDBC as follows:

1. **Type 1: JDBC/ODBC**—These require an ODBC (Open Database Connectivity) driver for the database to be installed. This type of driver works by translating the submitted queries into equivalent ODBC queries and forwards them via native API calls directly to the ODBC driver. It provides no host redirection capability.
2. **Type2: Native API (partly-Java driver)**—This type of driver uses a vendor-specific driver or database API to interact with the database. An example of such an API is Oracle OCI (Oracle Call Interface). It also provides no host redirection.
3. **Type 3: Open Protocol-Net**—This is not vendor specific and works by forwarding database requests to a remote database source using a net server component. How the net server component accesses the database is transparent to the client. The client driver communicates

with the net server using a database-independent protocol and the net server translates this protocol into database calls. This type of driver can access any database.

4. **Type 4: Proprietary Protocol-Net(pure Java driver)**—This has a same configuration as a type 3 driver but uses a wire protocol specific to a particular vendor and hence can access only that vendor's database. Again this is all transparent to the client.

Note: Type 4 JDBC driver is most preferred kind of approach in JDBC and is the fastest driver because it converts the jdbc calls into vendor specific protocol calls and it directly interacts with the database.

Difference between TYPE_SCROLL_INSENSITIVE and TYPE_SCROLL_SENSITIVE

You will get a scrollable ResultSet object if you specify one of these ResultSet constants.The difference between the two has to do with whether a result set reflects changes that are made to it while it is open and whether certain methods can be called to detect these changes. Generally speaking, a result set that is TYPE_SCROLL_INSENSITIVE does not reflect changes made while it is still open and one that is TYPE_SCROLL_SENSITIVE does. All three types of result sets will make changes visible if they are closed and then reopened.

Connection pooling

Connection pooling is a technique used for sharing server resources among requesting clients. Connection pooling increases the performance of Web applications by reusing active database connections instead of creating a new connection with every request. Connection pool manager maintains a pool of open database connections.

- you gets a reference to the pool
- you gets a free connection from the pool
- you performs your different tasks
- you frees the connection to the pool

JDBC Best Practices: These are few points to consider:

- ✓ Use a connection pool mechanism whenever possible.
- ✓ Use prepared statements. These can be beneficial, for example with DB specific escaping, even when used only once.
- ✓ Use stored procedures when they can be created in a standard manner. Do watch out for DB specific SP definitions that can cause migration headaches.
- ✓ Even though the jdbc promotes portability, true portability comes from NOT depending on any database specific data types, functions and so on.
- ✓ Select only required columns rather than using select * from Tablexyz.
- ✓ Always close Statement and ResultSet objects as soon as possible.
- ✓ Write modular classes to handle database interaction specifics.
- ✓ Work with DatabaseMetaData to get information about database functionality.
- ✓ Softcode database specific parameters with, for example, properties files.
- ✓ Always catch AND handle database warnings and exceptions. Be sure to check for additional pending exceptions.
- ✓ Test your code with debug statements to determine the time it takes to execute your query and so on to help in tuning your code. Also use query plan functionality if available.
- ✓ Use proper (and a single standard if possible) formats, especially for dates.
- ✓ Use proper data types for specific kind of data. For example, store birthdate as a date type rather than, say, varchar.

DDL: Data Definition Language statements are used to define the database structure or schema. Some examples:

CREATE - to create objects in the database
ALTER - alters the structure of the database
DROP - delete objects from the database
TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
COMMENT - add comments to the data dictionary
RENAME - rename an object

DML: Data Manipulation Language statements are used for managing data within schema objects. Some examples:

SELECT - retrieve data from the a database
INSERT - insert data into a table
UPDATE - updates existing data within a table
DELETE - deletes all records from a table, the space for the records remain
MERGE - UPSERT operation (insert or update)
CALL - call a PL/SQL or Java subprogram
EXPLAIN PLAN - explain access path to data
LOCK TABLE - control concurrency

DCL: Data Control Language

GRANT - gives user's access privileges to database
REVOKE - withdraw access privileges given with the GRANT command

TCL: Transaction Control (TCL) statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

COMMIT - save work done
SAVEPOINT - identify a point in a transaction to which you can later roll back
ROLLBACK - restore database to original since the last COMMIT
SET TRANSACTION - Change transaction options like isolation level and what rollback segment to use

Transactions

– By default, after each SQL statement is executed the changes are automatically committed to the database
– Turn auto-commit off to group two or more statements together into a transaction connection.setAutoCommit(false)
– Call commit to permanently record the changes to the database after executing a group of statements
– Call rollback if an error occurs

Useful Connection Methods (for Transactions)

• getAutoCommit/setAutoCommit
  – By default, a connection is set to auto-commit
  – Retrieves or sets the auto-commit mode
• commit
  – Force all changes since the last call to commit to become permanent
  – Any database locks currently held by this Connection object are released
• rollback
  – Drops all changes since the previous call to commit
  – Releases any database locks held by this Connection object

JDBC Transaction ACID Properties

A Transaction is a unit of work performed on the database and treated in a reliable way independent of other transaction. In database transaction processing ACID property refers to the Atomicity, Consistency, Isolation, Durability respectively.

- ✓ **Atomicity**- This property says that all the changes to the data is performed if they are single operation. For example suppose in a bank application if a fund transfer from one account to another account the atomicity property ensures that is a debit is made successfully in one account the corresponding credit would be made in other account.
- ✓ **Consistency**- The consistency property of transaction says that the data remains in the consistence state when the transaction starts and ends. for example suppose in the same bank account, the fund transfer from one account to another account, the consistency property ensures that the total value(sum of both account) value remains the same after the transaction ends.

- ✓ **Isolation**- This property says that, the intermediate state of transaction are hidden/ invisible to another transaction process. Suppose in the the bank application, the isolation property ensures that the fund transfer from one account to another account, the transaction sees the fund transfer in one account or the other account.
- ✓ **Durability**- The Durability says that when the transaction is completed successfully, the changing to the data is persist and is not un-done, even in the event of system failure.

Dirty Read:
Quite often in database processing, we come across the situation wherein one transaction can change a value, and a second transaction can read this value before the original change has been committed or rolled back. This is known as a dirty read scenario because there is always the possibility that the first transaction may rollback the change, resulting in the second transaction having read an invalid value.
While you can easily command a database to disallow dirty reads, this usually degrades the performance of your application due to the increased locking overhead. Disallowing dirty reads also leads to decreased system concurrency.

Non-Repeatable read:
One of the ISO-ANSI SQL defined "phenomena" that can occur with concurrent transactions. If one transaction reads a row, then another transaction updates or deletes the row and commits, the first transaction, on re-read, gets modified data or no data. This is an inconsistency problem within a transaction and addressed by isolation levels.

Phantom Read
A "phantom" read occurs when one transaction reads all rows that satisfy a WHERE condition, and a second transaction inserts a row that satisfies that WHERE condition, the first transaction then rereads for the same condition, retrieving the additional "phantom" row in the second read. (from book: JDBC Recipes A Problem-Solution Approach)

Transaction Isolation Levels

```
Connection.setTransactionIsolation (level)
```

JDBC Defined Constant	Description
TRANSACTION_READ_UNCOMMITTED	Allows dirty reads, non-repeatable reads, and phantom reads to occur.
TRANSACTION_READ_COMMITTED	Ensures only committed data can be read.
TRANSACTION_REPEATABLE_READ	Is close to being "serializable," however, "phantom" reads are possible.
TRANSACTION_SERIALIZABLE	Dirty reads, non-repeatable reads, and phantom reads are prevented. Serializable.

In addition, JDBC defines an additional constant, TRANSACTION_NONE, which is used to indicate that the driver does not support transactions.

How can I determine the isolation levels supported by my DBMS?

```
DatabaseMetaData.supportsTransactionIsolationLevel(int level).
```

Concurrency issues with JDBC
JDBC is based on ISO-ANSI SQL, which provides for database concurrency using isolation levels. These are defined to deal with the "phonema" of dirty reads, non-repeatable reads and phantom inserts. "The four isolation levels guarantee that each SQL-transaction will be executed completely or not at all, and that no updates will be lost." While almost all programs will be concerned with dirty reads, in the real world we probably want to use and see the most current data, so non-repeatable reads and phantom inserts may not be an issue. Here's a table that briefly summarizes what the defined isolation levels guarantee:

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Insert
Read Uncommitted	Possible	Possible	Possible
Read Committed	Not Possible	Possible	Possible

Repeatable Read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	NotPossible

In JDBC, the isolation level is set by Connection.setTransactionIsolation(int level) with the default being auto-commit. Note that a particular database may not support all isolation levels (DatabaseMetaData.supportsTransactionIsolationLevel(int) will validate a particular isolation level.)

Also, isolation level is a Connection property; this means that it applies to all Statements created by a Connection, and that a commit or rollback also applies to all Statements.

Finally, isolation levels are more concerned with data integrity in the face of concurrent transactions, than with allowing concurrent access for many users of the database.

What are the considerations for deciding on transaction boundaries?

Transaction processing should always deal with more than one statement and a transaction is often described as a Logical Unit of Work (LUW). The rationale for transactions is that you want to know definitively that all or none of the LUW completed successfully. Note that this automatically gives you restart capability.

Typically, there are two conditions under which you would want to use transactions:

- Multiple statements involving a single file - An example would be inserting all of a group of rows or all price updates for a given date. You want all of these to take effect at the same time; inserting or changing some subset is not acceptable.
- Multiple statements involving multiple files - The classic example is transferring money from one account to another or double entry accounting; you don't want the debit to succeed and the credit to fail because money or important records will be lost. Another example is a master/detail relationship, where, say, the master contains a total column. If the entire LUW, writing the detail row and updating the master row, is not completed successfully, you A) want to know that the transaction was unsuccessful and B) that a portion of the transaction was not lost or dangling. Therefore, determining what completes the transaction or LUW should be the deciding factor for transaction boundaries.

Database cursors

A cursor is actually always on the database server side. When you execute an SQL SELECT and create a ResultSet in JDBC, the RDBMS creates a cursor in response. When created, the cursor usually takes up temporary memory space of some sort inside the database.

What is a JDBC 2.0 DataSource?

The DataSource class was introduced in the JDBC 2.0 Optional Package as an easier, more generic means of obtaining a Connection. The actual driver providing services is defined to the DataSource outside the application (Of course, a production quality app can and should provide this information outside the app anyway, usually with properties files or ResourceBundles). The documentation expresses the view that DataSource will replace the common DriverManager method.

What is Metadata and why should I use it?

Metadata ('data about data') is information about one of two things:
1. Database information (java.sql.DatabaseMetaData), or
2. Information about a specific ResultSet (java.sql.ResultSetMetaData).

Use DatabaseMetaData to find information about your database, such as its capabilities and structure. Use ResultSetMetaData to find information about the results of an SQL query, such as size and types of columns.

Servlets

Section 1: The Servlet Model

1.1 doGet, doPost, doPut. Parms are: (HttpServletRequest req, HttpServletResponse res)

1.2 GET, POST, HEAD – what triggers cause a browser to do this??

- ✓ GET – user enters url into browser.
  <FORM METHOD="GET" ACTION="servlet_name">
- ✓ POST – user fills in html form and clicks submit, form uses POST method to pass form values.
  <FORM METHOD="POST" ACTION="servlet_name">
- ✓ HEAD – request only page headers. A way to check to see if a document has been updated since the last request.
  <FORM METHOD="HEAD" ACTION="servlet_name">

1.3

a) HttpServletRequest request.getParameters() to get list of parameters.
b) ServletConfig.getInitParameter()
c) HttpservletRequest.getHeader(String headerName) : String – other methods:
getHeaderNames(), getHeaders : Enumeration.
d) General purpose HttpServletResponse.setHeader(name, value). Also, can use
setDateHeader(String, long) and setIntHeader(String, int).
ServletResponse.setContentType(String) "text/html".
e) PrintWriter out = response.getWriter();
f) OutputStream out = resonse.getOutputStream()
g) HttpServletResponse.sendRedirect(String url)

1.4
Request – ServletRequest.setAttribute(String, Object), deleteAttribute(String), getAttribute(String), getAttributeNames().

Session - HttpSession.setAttribute(String, Object), deleteAttribute(String), getAttribute(String), getAttributeNames().

Context – ServletContext.getAttribute(String), getAttributeNames(), removeAttribute(name), setAttribute(name)

1.5
init – call when first created and not called again for each user request. One time setup code goes here. Servlet container controls life-cycle.

service – Server receives request and spawns separate thread for each user request.
Invokes service method which, by default, hand's off to doXXX methods doGet, doPost

destroy – executed prior to server unloading the servlet. Good for closing db connections, etc.

1.6 RequestDispatcher -

```
RequestDispatcher rd;
ServletContext sc = getServletContext();
rd = sc.getRequestDispatcher("/mainmenu.jsp");
rd.forward(req,res);
```

void	`forward(ServletRequest request, ServletResponse response)` Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
void	`include(ServletRequest request, ServletResponse response)` Includes the content of a resource (servlet, JSP page, HTML file) in the response.

Section 2: The Structure and Deployment of Modern Servlet Web Applications

2.1 Web Application Structure

\	Jsp, html, images (optional)
\WEB-INF	Web.xml, tld files, other xml config files.
\WEB-INF \classes*	Compiled classes
\WEB-INF \lib*.jar	Jarred classes

NOTE: Enters in WEB-INF and below are not visible outside the container. Root is visible.

2.2

a) Servlet instance – The servlet element contains the declarative data of a servlet

```
<!ELEMENT servlet (icon?, servlet-name, display-name?, description?, (servlet-class|jsp-file), init-param*, load-on-startup?, security-role-ref*)>
```

b) Servlet name – logical name used within dd

```
<!ELEMENT servlet-name (#PCDATA)>
```

c) Servlet class – fully qualified package name of servlet class.

```
<!ELEMENT servletclass (#PCDATA)>
```

d) Initialization parms – init parms for servlet.

```
<!ELEMENT init-param (param-name, param-value, description?)>
```

e) URL to named servlet mapping - map logical servlet name to URL that server will listen for.

```
<!ELEMENT servlet-mapping (servlet-name, url-pattern)>
```

Section 3: The Servlet Container Model

Application Events

Event Listeners instantiation and registration is done by the Web Container and must be declared in the web.xml deployment descriptor.

3.1 Interfaces, Uses, and Methods

a) ServletContext.getInitParameter(), getInitParameterNames(). These are specified in the web.xml.
b) Servlet context listeners are used to manage resources or state held at a VM level for the application. ServletContextListener.contextDestroyed() – server shut down. ContextInitialized – server up and ready for requests.
c) When attributes are added to the application environment context. ServletContextAttributeListener attributeAdded(), attributeRemoved(), attributeReplaced()
d) Fine grained control over contents of a Session. HTTPSessionAttriuteListener ServletContextAttributeListener attributeAdded(), attributeRemoved(), attributeReplaced()

3.2 Corresponding Deployment Descriptor Elements

a) Element: <context-param> Subelements: <param-name> <paramvalue> b, c, d)

```
<listener>
      <listener-class> com.something.classname</listener-class>
</listener>
Session & Context dd declaration is the same)
```

3.3 A Web Application marked as Distributable – servlet container to distribute to multiple JVM's for clustering,scalability, and failover.

- ✓ The context exists locally in the VM in which they were created and placed. This prevents the ServletContext from being used as a distributed shared memory store, Instead place it in a Session, DB, or EJB.
- ✓ Session is scoped to vm servicing session requests.
- ✓ Context is scoped to the web container's vm.
- ✓ Container is not required to propogate context or session events to other vm's.
  - o each container will have it's own separate copy of this.
  - o each container will have it's own and events are not propogated to the other.
  - o same as b

- each vm will have it's own and events are not propogated to the other.

Section 4: Designing and Developing Servlets to Handle Server-side Expectations

4.1 sendError, setStatus

- ✓ HTTPServletResponse.sendError(int), (int, string). Sends HTTP Error code to browser. If an error-page declaration has been made for the web application corresponding to the status code passed in, it will be served back in preference to the suggested msg parameter.
- ✓ HTTPServletResponse.setStatus(int), set HTTP status code when there is an error, does not invoked web app error page.

4.2 Deployment Descriptor for specifying an error page for the web application for a given HTTP error code or java exception.

```
<error-page>
   <error-code>404</error-code>
<location>/404.html</location>
</error-page>
```

OR

```
<exception-type>java.lang.NullPointerException</exception-type>
```

Instead of <error-code>

Request Dispatcher could be used to manually forward a request to an error page so long as the request attributes (status_code, exception, request_uri, servlet_name) are contained in the request.

4.3 Servlet/WebApp Log File ServletContext (interface), GenericServlet methods:

void	`log(java.lang.String msg)` Writes the specified message to a servlet log file, usually an event log.
void	`log(java.lang.String message, java.lang.Throwable throwable)` Writes an explanatory message and a stack trace for a given Throwable exception to the servlet log file.

Section 5: Designing and Developing Servlets Using Session Management

5.1

a) HTTPServletRequest.getSession() : HTTPSession
b) HTTPSession.setAttribute(name, object) (putValue is deprecated)
c) HTTPSession.getAttribute(name) : Object
d) HTTPSessionAttributeListener attributeAdded(HTTPSessionBindingEvent), attributeRemoved(evt), attributeReplaced(evt)
e) HTTPSessionListener.sessionCreated(HTTPSessionEvent), sessionDestroyed(evt)
f) HTTPSession.invalidate()

5.2 Invalidate Session. A session will be invalidated when the session times out (container/server invalidates) or is invalidated in code by the use of HTTPSession.invalidate().

5.3 URL Rewriting
URL rewriting is the lowest common denominator of session tracking. When a client will not accept a cookie, URL rewriting may be used by the server as the basis for session tracking. URL rewriting involves adding data, a session id, to the URL path that is interpreted by the container to associate the request with a session. The session id must be encoded as a path parameter in the URL string.

The name of the parameter must be *jsessionid*. Here is an example of a URL containing encoded path information:

```
http://www.myserver.com/catalog/index.html;jsessionid=1234
```

Section 6: Designing and Developing Secure Web Applications

6.1
a) authentication – verifyuser is who they claim to be. See 6.3.
b) authorization – authenticated user is allowed to access a particular resource.
c) data integrity – the means to prove that information has not been modified by a third party while in transit.
d) Auditing – record a permanent log of user activities within the application.
e) malicious code – Code aimed at breaking the security rules of a web application (buffer overflow mainly).
f) web site attacks – external attacks by hackers to the web application.

6.2 a) Security Constraint Example

```
<security-constraint>
        <webresource-collection>
                <web-resource-name>Entire application</web-resource-name>
                <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
                <role-name>manager</role-name>
        </auth-constraint>
</security-constraint>
```

Web resource – A web resource collection is a set of URL patterns and HTTP methods that describe a set of resources to be protected. All requests that contain a request path that matches a URL pattern described in the web resource collection is subject to the constraint. The container matches URL patterns defined in security constraints using the same algorithm described in this specification for matching client

b) requests to servlets and static resources.

```
<web-resource-collection>
  <web-resource-name>SalesInfo</web-resource-name>
  <url-pattern>/salesinfo/*</url-pattern>
  <http-method>GET</http-method>
  <http-method>POST</http-method>
</web-resource-collection>
```

c) Determine what kind of user authentication is required

```
<login-config>
       <auth-method>BASIC|DIGEST|FORM</auth-method>
      <realm-name>HTTP Realm Name </realm-name>
</login-config>
```

d) Determine security roles

```
<security-role-ref>
  <role-name>XYZ</role-name>
   <role-link>customer service</role-link>
</security-role-ref>
…
<security-role>
  <role-name>administrator</role-name>
  <role-name>customerservice</role-name>
</security-role>
```

6.3 Authentication Types

BASIC – HTTP Basic Authentication – Web server/container prompts web client for username/pwd. User sends Base64 encoded string. Brower accomplishes this by by popping up a dialog. Browser caches authentication info for subsequent requests.

DIGEST – HTTP Digest Authentication – More secure form of BASIC authentication. Browser sends MD-5 of username/pwd as, url, & http method. Only IE 5.x supports this form of authentication. Not required in spec.

FORM – Form Based Authentication – Developer creats custom login/error screens.

Server manages the display of these. Form action: j_security_check, parms: j_username, j_password. Must be configured in DD:

```
<form-login-config>
   <form-login-page>/loginpage.jsp</form-login-page>
   <form-error-page>/errorpage.jsp</form-error-page>
</form-login-config>
```

CLIENT-CERT – HTTPS Client Authentication – Uses SSL. Uses public-key encryption to establish a secure session to transmit data between client and server. Authenticate server and optionally the client. Provide encrypted connection to exchange messages.

Section 7: Designing and Devloping Thread-safe Servlets

7.1 Thread Safety Issues

a) Y
b) N
c) N
d) Y – only within the request thread
e) N
f) N

7.2 Single Thread Model guarantees one thread will execute through a servlet's service method at a time. Objects (ie. Session) may be accessible by multiple instances however. Multiple Thread Model usually has a single instance to handle all requests.
A container with Single Thread Model may instantiate multiple servlet instances to handle the load.

7.3 Implementing SingleThreadModel interface marks the servlet (to the container) as being capable of only executing one request at a time.

Servlet Complete Reference

1. The Servlet Model

1.1. Identify corresponding method in HttpServlet class for each of the followingHTTP methods:
- 1.1.1. GET: protected void **doGet**(HttpServletRequest req, HttpServletResponse res) throws ServletException,IOException
- 1.1.2. POST:protected void **doPost**(HttpServletRequest req,HttpServletResponse res) throws ServletException, IOException
- 1.1.3. PUT:protected void **doPut**(HttpServletRequest req,HttpServletResponse res) throws ServletException, IOException

1.2. GET, POST and HEAD
- 1.2.1. Identify triggers that might cause a browser to use:
  - 1.2.1.1. GET: (a) typing url directly into a browser, (b) clicking on a hyperlink, (c)submitting html form with 'method=get' or no method attribute
  - 1.2.1.2. POST:(a) submitting html form with 'method=post'
  - 1.2.1.3. HEAD:(a) may be used by a browser to check modification time for purposes of caching
- 1.2.2. Identify benefits or functionality of:
  - 1.2.2.1. GET:
    - 1.2.2.1.1. designed for getting information (e.g. document, chart, results of query)
    - 1.2.2.1.2. caninclude a query string (some servers limit this to about 240 characters) forsending information to server
    - 1.2.2.1.3. requested page can be bookmarked
  - 1.2.2.2. POST:
    - 1.2.2.2.1. designed for posting information (e.g. credit card #, info to be stored in a db)
    - 1.2.2.2.2. passesall its data (of unlimited length) to server as part of its http request body
    - 1.2.2.2.3. postscannot be bookmarked or, in some cases, even reloaded
    - 1.2.2.2.4. hidessensitive information from server log by including it in the message bodyinstead of the url query string
  - 1.2.2.3. HEAD:
    - 1.2.2.3.1. sent by a client when it wants to see only the headers of the response, todetermine the document's size, modification time, or generalavailability.
    - 1.2.2.3.2. Theservice() method treats HEAD requests specially. It calls doGet with a modifiedresponse object, which suppresses any output but retains headers.

1.3. For each of the following operations, identify the interface and method namethat should be used:
- 1.3.1. Retrieve HTML form parameters from the request:
  - 1.3.1.1. Enumeration ServletRequest.**getParameterNames**() - returns empty enum if no parameters
  - 1.3.1.2. StringServletRequest.**getParameter**(String name) - returns nullif does not exist
  - 1.3.1.3. String[]ServletRequest.**getParameterValues**(String name) - returnsnull if does not exist
- 1.3.2. Retrieve a servlet initialization parameter:
  - 1.3.2.1. Enumeration ServletConfig.**getInitParameterNames**() - returns empty enum if no init parameters
  - 1.3.2.2. StringServletConfig.**getInitParameter**(String name) - returnsnull if does not exist
- 1.3.3. Retrieve HTTP request header information:
  - 1.3.3.1. Enumeration HttpServletRequest.**getHeaderNames**() - returns empty enum if no headers
  - 1.3.3.2. StringHttpServletRequest.**getHeader**(String name) - returns nullif does not exist
  - 1.3.3.3. Enumeration HttpServletRequest.**getHeaders**(String name) - returns empty enum if no headers
  - 1.3.3.4. longgetDateHeader(String name) - returns -1 if does not exist
  - 1.3.3.5. intgetIntHeader(String name) - returns -1 if does not exist
- 1.3.4. Set an HTTP response header; set the content type of the response
  - 1.3.4.1. void HttpServletResponse.**setHeader**(String name, String value) - if header already exists, overwrites its value
  - 1.3.4.2. voidHttpServletResponse.**setIntHeader**(String name, int value)
  - 1.3.4.3. voidHttpServletResponse.**setDateHeader**(String name, longdate)
  - 1.3.4.4. voidHttpServletResponse.**addHeader**(String name, String value)- if header already exists, adds an additional value
  - 1.3.4.5. voidHttpServletResponse.**addIntHeader**(String name, int value)
  - 1.3.4.6. voidHttpServletResponse.**addDateHeader**(String name, longdate)
  - 1.3.4.7. voidHttpServletResponse.**setContentType**(String type) –if calling getWriter(), then setContentType should be called first
- 1.3.5. Acquire a text stream for the response
  - 1.3.5.1. PrintWriter ServletResponse.**getWriter**() throws IOException - character encoding may be set bycalling setContentType, which must be called before calling getWriter()
- 1.3.6. Acquire a binary stream for the response
  - 1.3.6.1. ServletOutputStream ServletResponse.**getOutputStream**() throws IOException
- 1.3.7. Redirect an HTTP request to another URL
  - 1.3.7.1. void HttpServletResponse.**sendRedirect**(String location) throws IllegalStateException IOException
  - 1.3.7.2. setsstatus to SC_MOVED_TEMPORARILY, sets the Location header, and performs an implicit reset on theresponse buffer before generating the redirect page; headers set beforesendRedirect() remain set
  - 1.3.7.3. must becalled before response body is committed, else throws IllegalStateException

1.3.7.4. the pathmay be relative or absolute
1.3.7.5. tosupport clients without redirect capability, method writes a short response bodythat contains a hyperlink to the new location; so do not write your own msg body

1.4. Identify the interface and method to access values and resources and to setobject attributes within the following three Web scopes:
1.4.1. Request (Interfaces: **ServletRequest andHttpServletRequest**)
1.4.1.1. Enumeration ServletRequest.**getAttributeNames**() - returns empty enumeration if no attributes
1.4.1.2. ObjectServletRequest.**getAttribute**(String name) - returns nullif does not exist
1.4.1.3. void**setAttribute**(String name, Object obj) - most often usedin conjunction with RequestDispatcher; attrib names should follow sameconvention as pkg names
1.4.1.4. void**removeAttribute**(String name)
1.4.1.5. StringServletRequest.**getCharacterEncoding**() - returns encodingused in request body, or null if not specified
1.4.1.6. intServletRequest.**getContentLength**() - returns length ofrequest body or -1 if unknown
1.4.1.7. StringServletRequest.**getContentType**() - returns mime type ofrequest body or null if unknown
1.4.1.8. StringServletRequest.**getProtocol**() - returns protocol/version,e.g. HTTP/1.1
1.4.1.9. StringServletRequest.**getScheme**() - scheme used to make thisrequest, e.g. ftp, http, https
1.4.1.10. StringServletRequest.**getServerName**()
1.4.1.11. intServletRequest.**getServerPort**()
1.4.1.12. StringHttpServletRequest.**getAuthType**() - e.g. BASIC, SSL, ornull if not protected
1.4.1.13. StringHttpServletRequest.**getContextPath**() - e.g."/myservlet"
1.4.1.14. StringHttpServletRequest.**getMethod**() - e.g. GET, POST, HEAD,PUT
1.4.1.15. StringHttpServletRequest.**getPathInfo**() - returns extra pathinfo (string following servlet path but preceding query string); null if doesnot exist
1.4.1.16. StringHttpServletRequest.**getPathTranslated**() - translatesextra path info to a real path on the server
1.4.1.17. StringHttpServletRequest.**getQueryString**() - returns querystring; null if does not exist
1.4.1.18. StringHttpServletRequest.**getRemoteUser**() - returns null ifuser not authenticated
1.4.1.19. Principal HttpServletRequest.**getUserPrincipal**() - returns null if user not authenticated
1.4.1.20. StringHttpServletRequest.**getRequestURI**() - e.g. if request is"POST /some/path.html HTTP/1.1", then returns "/some/path.html"
1.4.1.21. StringHttpServletRequest.**getServletPath**() - returns servletpath and name, but no extra path info
1.4.1.22. HttpSession HttpServletRequest.**getSession**(boolean create)
1.4.1.23. HttpSession HttpServletRequest.**getSession**() - calls getSession(true)
1.4.2. Session (Interface: **HttpSession**)
1.4.2.1. Enumeration HttpSession.**getAttributeNames**() - returns empty enumeration if no attributes; IllegalStateException ifsession invalidated
1.4.2.2. ObjectHttpSession.**getAttribute**(String name) - returns null ifno such object
1.4.2.3. voidHttpSession.**setAttribute**(java.lang.String name,java.lang.Object value)
1.4.2.4. voidHttpSession.**removeAttribute**(java.lang.String name)
1.4.2.5. StringHttpSession.**getId**() - returns unique session identifierassigned by servlet container
1.4.2.6. longHttpSession.**getLastAccessedTime**() - time when clientlast sent a request associated with this session
1.4.2.7. intHttpSession.**getMaxInactiveInterval**() - returns number ofseconds this session remains open between client requests; -1 if session shouldnever expire
1.4.2.8. voidHttpSession.**setMaxInactiveInterval**(int interval)
1.4.3. Context (Interface: **ServletContext**)
1.4.3.1. Enumeration **getAttributeNames**() - Returns an Enumeration containing the attribute names available within thisservlet context.
1.4.3.2. Object**getAttribute**(String name) - Returns the servletcontainer attribute with the given name, or null if there is no attribute bythat name.
1.4.3.3. void**setAttribute**(String name, java.lang.Object object) -Binds an object to a given attribute name in this servlet context.
1.4.3.4. void**removeAttribute**(String name) - Removes the attributewith the given name from the servlet context.
1.4.3.5. ServletContext **getContext**(String uripath) -Returns a ServletContext object that corresponds to a specified URL on theserver.
1.4.3.6. String**getInitParameter**(String name) - Returns a Stringcontaining the value of the named context-wide initialization parameter, or nullif does not exist.
1.4.3.7. Enumeration **getInitParameterNames**() -Returns names of the context's initialization parameters as Enumeration ofString objects
1.4.3.8. int**getMajorVersion**() - Returns the major version of theJava Servlet API that this servlet container supports.
1.4.3.9. int**getMinorVersion**() - Returns the minor version of theServlet API that this servlet container supports.
1.4.3.10. String**getMimeType**(String file) - Returns the MIME type of thespecified file, or null if the MIME type is not known.
1.4.3.11. RequestDispatcher **getNamedDispatcher**(String name) - Returns a RequestDispatcher object that acts as a wrapper forthe named servlet.
1.4.3.12. RequestDispatcher **getRequestDispatcher**(String path) - Returns a RequestDispatcher object that acts as a wrapper forthe resource located at the given path.
1.4.3.13. String**getRealPath**(String path) - Returns a String containingthe real path for a given virtual path.

1.4.3.14. java.net.URL **getResource**(String path) -Returns a URL to the resource that is mapped to a specified path.
1.4.3.15. InputStream **getResourceAsStream**(Stringpath) - Returns the resource located at the named path as an InputStream object.
1.4.3.16. String**getServerInfo**() - Returns the name and version of theservlet container on which the servlet is running.

1.5. For each of the following life-cycle method, identify its purpose and how andwhen it is invoked:
1.5.1. public void init() throws ServletException:
1.5.1.1. called after server constructs the servlet instance and before the serverhandles any requests
1.5.1.2. depending on the server and web app configuration, init() may be called at any ofthese times: (a) when server starts, (b) when the servlet is first requested,just before the service() method is invoked, (c) at the request of the serveradministrator
1.5.1.3. ifservlet specifies <load-on-startup/> in its web.xml file, then upon serverstartup, the server will create an instance of the servlet and call its init()method.
1.5.1.4. typically used to perform servlet initialization, e.g. loadingobjects used by servlet to handle requests, reading in servlet init parameters,starting a background thread.
1.5.1.5. servletcannot be placed into service if init method throws ServletException or does notreturn within a server-defined time period
1.5.1.6. init()can only be called once per servlet instance
1.5.2. public void service() throws ServletException, IOException:
1.5.2.1. called by the servlet container to allow the servlet to respond to a request.
1.5.2.2. thismethod is only called after the servlet's init() method has completedsuccessfully.
1.5.2.3. servlets typically run inside multithreaded servlet containers that can handlemultiple requests concurrently. developers must be aware to synchronize accessto any shared resources such as files and network connections, as well as the servlet's class and instance variables.
1.5.3. public void destroy():
1.5.3.1. called after the servlet has been taken out of service and all pending requeststo the servlet have been completed or timed out
1.5.3.2. givesthe servlet an opportunity to clean up any resources that are being held (forexample, memory, file handles, threads) and make sure that any persistent stateis synchronized with the servlet's current state in memory
1.5.3.3. callingsuper.destroy() causes GenericServlet.destroy() to write a note to the log thatthe servlet is being destroyed
1.5.3.4. destroy()called once per servlet instance; destroy() not called ifserver crashes, so should save state (if needed) periodically after servicingrequests
1.5.4. **Note**: servlet reloading
1.5.4.1. most servers automatically reload a servlet after its class file (under servletdir, e.g. WEB-INF/classes) changes. when a server dispatches a request to aservlet, it first checks whether the servlet's class file has changed on disk. If it has, then the server creates a new custom class loader, andreloads the entire web application context.
1.5.4.2. classreloading is not based on support class changes or on changes in classes foundin the server's classpath, which are loaded by the core, primordial classloader.

1.6. Use a RequestDispatcher to include or forward to a Web resource
1.6.1. include:
1.6.1.1. public void include(ServletRequest request, ServletResponse response) throwsServletException, IOException
1.6.1.2. Includesthe content of a resource (servlet, JSP page, HTML file) in the response. Inessence, this method enables programmatic server-side includes.
1.6.1.3. TheServletRequest object has its path elements (e.g. attributes request_uri,context_path, and servlet_path) and parameters remain unchanged from thecaller's.
1.6.1.4. Theincluded servlet cannot change the response status code or set headers; anyattempt to make a change is ignored.
1.6.1.5. Therequest and response parameters must be the same objects as were passed to thecalling servlet's service method.
1.6.1.6. Theincluded resource must use the same output mechanism (e.g. PrintWriter orServletOutputStream) as the caller's
1.6.1.7. Information can be passed to target using attached query string orusing request attributes set with setAttribute() method.
1.6.2. forward:
1.6.2.1. public void forward(ServletRequest request, ServletResponse response) throwsServletException, IOException
1.6.2.2. Forwards a request from a servlet to another resource (servlet, JSP file, orHTML file) on the server. This method allows one servlet to do preliminaryprocessing of a request and another resource to generate the response. Theforwarding servlet generates no output, but may set headers.
1.6.2.3. TheServletRequest object has its path attributes adjusted to match the path of thetarget resource. Any new request parameters are added to the original.
1.6.2.4. forward() should be called before the response hasbeen committed to the client (before response body output has been flushed). Ifthe response already has been committed, this method throws

anIllegalStateException. Uncommitted output in the response buffer isautomatically cleared before the forward.
1.6.2.5. Therequest and response parameters must be the same objects as were passed to thecalling servlet's service method.
1.6.2.6. Information can be passed to target using attached query string orusing request attributes set with setAttribute() method.
1.6.2.7. forwarding to an html page containing relative url's included (e.g. <IMG> tags) is a bad idea, because forward() does not notifyclient about the directory from which the page is served, hence the links may bebroken. Instead, use sendRedirect().
1.6.3. **Note**: to get a request dispatcher object:
1.6.3.1. public RequestDispatcher ServletRequest.getRequestDispatcher(String path) - pathmay be relative, and cannot extend outside current servlet context
1.6.3.2. publicRequestDispatcher ServletContext.getNamedDispatcher(String name) - name is theregistered servlet name in web.xml file
1.6.3.3. publicRequestDispatcher ServletContext.getRequestDispatcher(String path) - acceptsonly absolute paths, and not relative paths

2. The Structure and Deployment of Modern Servlet Web Applications

2.1. Identify the following:
2.1.1. Structure of a Web Application
2.1.1.1. the following web application hierarchy is placed under a context root directorywithin the server's webapps directory (or something similar, depending onthe server) :
2.1.1.1.1. /[any files to be served to the client, e.g. index.html, images/banner.gif]
2.1.1.1.2. /WEB-INF/web.xml
2.1.1.1.3. /WEB-INF/lib/[any required jar files]
2.1.1.1.4. /WEB-INF/classes/[servlet and support class files in theirpackage hierarchies, e.g. com/mycorp/frontend/CorpServlet.class]
2.1.2. Structure of a Web Archive file
2.1.2.1. this is a JAR archive of the Web Application structure above; it just has a WARextension so that people and tools know to treat it differently
2.1.2.2. a WARfile can be placed in a server's webapps directory, and the server willextract it on startup
2.1.3. Name of Web App deployment descriptor: web.xml
2.1.4. Name ofdirectories where you place the following:
2.1.4.1. Web App deployment descriptor: see 2.1.1.2
2.1.4.2. Web Appclass file: see 2.1.1.4
2.1.4.3. Anyauxhiliary JAR files: see 2.1.1.3

2.2. Identify the purpose or functionality for each of the following deploymentdescriptor elements:
2.2.1. servlet instance:
2.2.1.1. <servlet> {servlet-name, servlet-class, init-param, etc.} </servlet>
2.2.1.2. declaresa servlet instance; included within <web-app> </web-app> tags
2.2.2. servlet name:
2.2.2.1. <servlet-name></servlet-name>
2.2.2.2. registers the servlet under a specific name
2.2.3. servlet class:
2.2.3.1. <servlet-class></servlet-class>
2.2.3.2. containsthe fully qualified class name of the servlet
2.2.4. initialization parameters:
2.2.4.1. <init-param> {param-name, param-value} </init-param>
2.2.4.2. definesvalues that can be set at deployment time and read at run-time viaServletConfig.getInitParameter(String name)
2.2.5. url to named servlet mapping
2.2.5.1. <servlet-mapping> <servlet-name> helloServlet </servlet-name> <url-pattern>/hello.html </url-pattern> </servlet-mapping>← this maps http://server:port/context_root/hello.html to the helloServletservlet.
2.2.5.2. zero ormore mappings may be defined per web app
2.2.5.3. 4 typesof mappings, searched in the following order: (a) explicitmappings, e.g. /hello.html (b) path prefix mappings e.g. /dbfile/* (c)extension mappings e.g. *.jsp or *.gif (d) the default mapping "/",identifying the default servlet for the web app

3. The Servlet Container Model

3.1. Servlet Context Init Parameters
3.1.1.1. purpose: defines init parameters accessible by all servlets in the webapplication context; set-able at deployment-time, but accessible at run-time
3.1.1.2. interfaces (or classes): javax.servlet.ServletContext
3.1.1.3. methods:public Enumeration getInitParameterNames() and public StringgetInitParameter(String name)
3.1.1.4. webappdeployment descriptor element name: <context-param> {param-name, param-value}</context-param>
3.1.1.5. behaviorin a distributable: Consider that a different instance of the ServletContext may exist on each different JVM and/or machine. Therefore, the context should not be used to store application state. Any stateshould be stored externally, e.g. in a database or ejb.

3.2. Servlet Context Listener
- 3.2.1.1. purpose: An object that implements the ServletContextListener interface is notifiedwhen its web app context is created or destroyed
- 3.2.1.2. interfaces (or classes): javax.servlet.ServletContextListener
- 3.2.1.3. methods:
  - 3.2.1.3.1. void contextInitialized(ServletContextEvent e): called during web server startupor when context is added or reloaded; requests will not be handled until thismethod returns
  - 3.2.1.3.2. voidcontextDestroyed(ServletContextEvent e): called during web server shutdown orwhen context is removed or reloaded; request handling will be stopped beforethis method is called
- 3.2.1.4. webapp deployment descriptor element name: <listener> <listener-class>{fully qualified class name} </listener-class> <listener>
- 3.2.1.5. behaviorin a distributable: Each context instance (on different jvm's and/or machines) will have its own instance of the listener object. Therefore,if a context on one jvm/machine is initialized or destroyed, it will not triggera listener on any other jvm/machine.

3.3. Servlet Context Attribute Listener
- 3.3.1.1. purpose: An object that implements the ServletContextAttributeListener interfaceis notified when attributes are added to or removed from its web app context
- 3.3.1.2. interfaces (or classes):javax.servlet.ServletContextAttributeListener
- 3.3.1.3. methods:voidattributeAdded/attributeRemoved/attributeReplaced(ServletContextAttributeEvente)
- 3.3.1.4. webappdeployment descriptor element name: <listener> <listener-class> {fullyqualified class name} </listener-class> <listener>
- 3.3.1.5. behaviorin a distributable: Addition, removal or replacement of an attribute in acontext will only affect the listener for that context, and not other context"instances" on other jvm's and/or machines.

3.4. HttpSession Attribute Listener
- 3.4.1.1. purpose: An object that implements the HttpSessionAttributeListener interface isnotified when a session attribute is added, removed or replaced
- 3.4.1.2. interfaces (or classes):javax.servlet.http.HttpSessionAttributeListener
- 3.4.1.3. methods: void attributeAdded/attributeRemoved/attributeReplaced(HttpSessionBindingEvente)
- 3.4.1.4. webappdeployment descriptor element name: <listener> <listener-class> {fullyqualified class name} </listener-class> <listener>
- 3.4.1.5. behaviorin a distributable: sessions may migrate from one jvm or machine to another;hence the session unbind event may occur on a different jvm/machine than thesession bind event.

3.5. Http Session Listener
- 3.5.1.1. purpose: An object that implementsthe HttpSessionListener interface is notified when a session is created ordestroyed in its web app context
- 3.5.1.2. interfaces (or classes): javax.servlet.http.HttpSessionListener
- 3.5.1.3. methods:
  - 3.5.1.3.1. void sessionCreated(HttpSessionEvent e)
  - 3.5.1.3.2. voidsessionDestroyed(HttpSessionEvent e) - called when session is destroyed(invalidated)
- 3.5.1.4. webapp deployment descriptor element name: <listener> <listener-class>{fully qualified class name} </listener-class> <listener>
- 3.5.1.5. behaviorin a distributable: sessions may migrate from one jvm or machine to another;hence the session destroy event may occur on a different jvm/machine than thesession create event.

4. Designing and Developing Servlets to Handle Server-Side Exceptions

4.1. For each of the following cases, identify correctly constructed code forhandling business logic exceptions, and match that code with correct statementsabout the code's behavior:
- 4.1.1. return an http error using setStatus
  - 4.1.1.1. public void HttpServletResponse.setStatus(int statusCode)
  - 4.1.1.2. if thisis not called, the server by default sets the status code to SC_OK(200).
  - 4.1.1.3. examplestatus codes: HttpServletResponse.SC_OK(200), SC_NOT_FOUND(404), SC_NO_CONTENT,SC_MOVED_TEMPORARILY/PERMANENTLY, SC_UNAUTHORIZED, SC_INTERNAL_SERVER_ERROR, SC_NOT_IMPLEMENTED,SC_SERVICE_UNAVAILABLE
  - 4.1.1.4. callingsetStatus() on an error leaves a servlet with the responsibility of generatingthe error page
  - 4.1.1.5. must becalled before the response is committed, otherwise call is **ignored**
- 4.1.2. return an http error using sendError
  - 4.1.2.1. public void HttpServletResponse.sendError(int statusCode[, StringstatusMessage]) throws IllegalStateException, IOException
  - 4.1.2.2. thesendError() method causes the server to generate and send an appropriateserver-specific page describing the error (unless <error-page> defined inweb.xml)
  - 4.1.2.3. with thetwo argument version of this method, the server may include the status messagein the error page, depending on the server implementation
  - 4.1.2.4. must becalled before response body is committed, else throws IllegalStateException

4.2. Given a set of business logic exceptions, identify the following:
- 4.2.1. configuring deployment descriptor for error handling

4.2.1.1. <web-app> … <error-page> <error-code> 404 </error-code> <location> /404.html </location></error-page> … </web-app>
4.2.1.2. thisspecifies that any call to sendError(), from within this web app, with 404 errorcode should display /404.html; this includes requests for static pages thatresult in 404 error code
4.2.1.3. thevalue of location must begin with '/', is treated as based in thecontext root, and must refer to a resource within the context
4.2.1.4. <location> may be dynamic (e.g. jsp, servlet); for these, theserver makes available the following request attributes:javax.servlet.error.status_code and javax.servlet.error.message
4.2.2. configuring deployment descriptor for exception handling
4.2.2.1. <web-app> … <error-page> <exception-type> javax.servlet.ServletException</exception-type> <location> /servlet/ErrorDisplay </location></error-page> … </web-app>
4.2.2.2. how theserver handles exceptions thrown by a servlet is server-dependent, unless an<error-page> entry exists for a specific exception type or a superclass
4.2.2.3. <location> may be dynamic (e.g. jsp, servlet); for these, the server makesavailable the following request attributes: javax.servlet.error.exception_type &javax.servlet.error.message; the exception object itself is not made available;hence no way to get a stack trace
4.2.2.4. servlets must catch all exceptions except those that subclass ServletException,IOException and RuntimeException (IOException may be caused by client closingthe socket by exiting the browser)
4.2.2.5. aServletException may be created with a message and a "root cause", bothoptional, e.g. { throw new ServletException("execution interrupted",InterruptedException); }
4.2.2.6. publicThrowable ServletException.getRootCause() returns the root cause exception
4.2.2.7. javax.servlet package also defines a subclass of ServletExceptioncalled UnavailableException(String msg[, int seconds]), which causes server totake servlet out of service
4.2.3. using RequestDispatcher to forward to an error page: see section 1.6above

4.3. Identify the method used for the following:
4.3.1. writing a message to the Web App log:
4.3.1.1. void log(String msg) - Writes the specified message to a servlet log file,usually an event log.
4.3.1.2. voidlog(String message, java.lang.Throwable throwable) - Writes an explanatorymessage and a stack trace for a given Throwable exception to the servlet logfile.
4.3.1.3. theseare methods are available in GenericServlet and ServletException
4.3.2. writing a message and an exception to the Web App log:
4.3.2.1. public void GenericServlet.log(String msg, Throwable t)
4.3.2.2. writesthe given message and the Throwable's stack trace to a servlet log; exactoutput format and location of log are server specific

5. Designing and Developing Servlets Using Session Management

5.1. Identify the interface and method for each of the following:
5.1.1. retrieve a session object across multiple requests to the same or differentservlets within the same webapp
5.1.1.1. public HttpSession HttpServletRequest.getSession([boolean create])
5.1.1.2. if noargument provided, then server will automatically create a new session object ifnone exists for the user in the web app context
5.1.1.3. to makesure the session is properly maintained, getSession must be called at least oncebefore committing the response
5.1.1.4. sessionsare scoped at the web application level; so a servlet running inside one contextcannot access session information saved by another context.
5.1.1.5. behindthe scenes, the client's session id is usually saved on the client in acookie called JSESSIONID. For client that don't support cookies, the session ID can be sent as part of a rewritten URL,encoded using a jsessionid path parameter.
5.1.1.6. notethat a requested session id may not match the id of the session returned by the getSession() method, such as when the id isinvalid. one can call req.isRequestedSessionIDValid() to test if the requestedsession id (that which was defined in the rewritten url or the persistentcookie) is valid.
5.1.2. store objects into a session object
5.1.2.1. public void HttpSession.setAttribute(String name, Object value) throwsIllegalStateException
5.1.2.2. bindsthe specified object under the specified name. Any existing binding with thesame name is replaced.
5.1.2.3. IllegalStateException thrown if session being accessed is invalid
5.1.3. retrieve objects from a session object
5.1.3.1. public Object HttpSession.getAttribute(String name) throws IllegalStateException-- returns the object bound under the specified name or null if there is nobinding
5.1.3.2. publicEnumeration HttpSession.getAttributeNames() throws IllegalStateException --returns all bound attribute names as an enumeration of Strings (empty enum if nobindings)
5.1.3.3. publicvoid HttpSession.removeAttribute(String name) throws IllegalStateException --removes binding or does nothing if binding does not exist
5.1.4. respond to the event when a particular object is added to a session
5.1.4.1. any object that implements the javax.servlet.http.HttpSessionBindingListener interface is notified when it is bound to or unbound from asession.
5.1.4.2. publicvoid valuBound(HttpSessionBindingEvent event) is called when the object is boundto a session
5.1.4.3. publicvoid valuUnbound(HttpSessionBindingEvent event) is called when the object isunbound from a session, by being removed or replaced, or by having the sessioninvalidated
5.1.5. respond to the event when a session is created or destroyed: seesection 3.5

5.1.6. expunge asession object
5.1.6.1. public void HttpSession.invalidate() – causes the session to be immediately invalidated. All objects stored in thesession are unbound. Call this method to implement a "logout".

5.2. given a scenario, state whether a session object will be invalidated
5.2.1. ideally, a session would be invalidated as soon as the user closed his browser,browsed to a different site, or stepped away from his desk. Unfortunately,there's no way for a server to detect any of these events.
5.2.2. sessionmay expire automatically, after a set timeout of inactivity (tomcat default is30 minutes)
5.2.3. timeoutcan be overridden in web.xml file by specifying<web-app>…<session-config><session-timeout>e.g.60</session-timeout></session-config> </web-app>
5.2.4. timeoutcan be overridden for a specific session by callingHttpSession.setMaxInactiveInterval(int secs) – negative value indicatessession should never time out.
5.2.5. sessionmay expire manually, when it is explicitly invalidated by a servlet by callinginvalidate()
5.2.6. a servershutdown may or may not invalidate a session, depending on the capabilities ofthe server
5.2.7. when asession expires (or is invalidated), the HttpSession object and the data valuesit contains are removed from the system; if you need to retain information beyond a sessionlifespan, you should keep it in an external location (e.g. a database)

5.3. given that url-rewriting must be used for session management, identify thedesign requirement on session-related html pages
5.3.1. For a servlet to support session tracking via URL rewriting, it has to rewriteevery local URL before sending it to the client.
5.3.2. publicString HttpServletResponse.encodeURL(String url)
5.3.3. publicString HttpServletResponse.encodeRedirectURL(String url)
5.3.4. both methods encode the given url to include the session id and returns the newurl, or, if encoding is not needed or is not supported, it leaves the urlunchanged. The rules for when and how to encode are server-specific.
5.3.5. note that when using session tracking based on url rewriting that multiplebrowser windows can belong to different sessions or the same session, dependingon how the windows were created and whether the link creating the windows wasurl rewritten.

5.4. **Note**: Using Cookies:
5.4.1.1. To send a cookie to a client: {Cookie cookie = newCookie("name","value"); res.addCookie(cookie);}.
5.4.1.2. Toretrieve cookies: {Cookie[] cookies = req.getCookies();}

5.5. **Note**: Http Session Activation Listener
5.5.1. purpose: Objects that are bound to a session may listen to container eventsnotifying them when that session will be passivated and when that session hasbeen activated. A container that migrates sessions between VMs or persistssessions is required tonotify all attributes bound to sessions implementingHttpSessionActivationListener.
5.5.2. voidsessionWillPassivate(HttpSessionEvent e) - session is about to move; it willalready be out of service when this method is called
5.5.3. voidsessionDidActivate(HttpSessionEvent e) - session has been activated on newserver; session will not yet be in service when this method is called

6. Designing and Developing Secure Web Applications

6.1. identify correct descriptions or statements about the security issues:
6.1.1. Authentication: Being able to verify the identities of the partiesinvolved
6.1.2. Authorization: Limiting access to resources to a select set of usersor programs
6.1.3. Integrity:Being able to verify that the content of the communication is not changed duringtransmission
6.1.4. Auditing: Keeping a record of resource access that was granted or denied mightbe useful for audit purposes later. To that end, auditing and logs serve theuseful purposes of preventing a break-in or analyzing a break-in post mortem.
6.1.5. MaliciousCode:
6.1.6. Web SiteAttacks:
6.1.7. Confidentiality: Ensuring that only the parties involved canunderstand the communication

6.2. Identify deployment descriptor element names, and their structures, that declarethe following
6.2.1. <web-app>
6.2.1.1. <servlet> <servlet-name>secretSalary<servlet-name><servlet-class>SalaryServer</servlet-class> </sevlet>
6.2.1.2. <**security-constraint**>← indicates certain pages in a web app are to be accessed by users in a certainrole (role-to-user mappings stored in server-specific format)
6.2.1.2.1. <**web-resource-collection**>
6.2.1.2.1.1. <web-resource-name>protectedResource</web-resource-name>
6.2.1.2.1.2. <url-pattern>/servlet/SalaryServer</url-pattern>← same wildcards allowed as forservlet mappings
6.2.1.2.1.3. <url-pattern>/servlet/secretSalary</url-pattern>
6.2.1.2.1.4. <http-method>GET</http-method>← if no methods specified, then all methods are protected
6.2.1.2.1.5. <http-method>POST</http-method>
6.2.1.2.2. </web-resource-collection>

6.2.1.2.3. <auth-constraint><role-name>manager</role-name><role-name>ceo</role-name></auth-constraint>← if no role-name, then not viewable by anyuser; if role-name = "*" then viewable by all roles
6.2.1.2.4. <user-data-constraint><transport-guarantee>CONFIDENTIAL</transport-guarantee></user-data-constraint>← optional, indicates SSL security
6.2.1.3. </security-constraint>
6.2.1.4. <**login-config**>
6.2.1.4.1. <auth-method>BASIC/DIGEST/FORM/CLIENT-CERT</auth-method>
6.2.1.4.2. <realm-name>Default </realm-name>← optional, only useful for BASIC authentication
6.2.1.4.3. <form-login-config> ← optional, only useful for FORM based authentication
6.2.1.4.3.1. <form-login-page>/loginpage.html</form-login-page><form-error-page>/errorpage.html</form-error-page>
6.2.1.4.4. </form-login-config>
6.2.1.5. </login-config>
6.2.1.6. <**security-role**><role-name>manager</role-name></security-role>← not req'd; explicitly declaring the webapp's roles supports tool-based manipulation of the file
6.2.2. </web-app>

6.3. For each of the following authentication types, identify the correct definition of itsmechanism
6.3.1. BASIC
6.3.1.1. web server maintains a db of usernames and passwords, and identifies certain webresources as protected. When these are accessed, web server requests username and password; this information is sent back to the server, whichchecks it against its database; and either allows or denies access.
6.3.1.2. Disadvantage: provides no confidentiality, no integrity, and onlythe most basic authentication.
6.3.1.3. Disadvantage: Transmitted passwords are encoded using easilyreversable Base64 encoding, unless additional SSL encryption employed.
6.3.1.4. Disadvantage: plus, passwords are often stored on server in cleartext
6.3.1.5. Advantage: very easy to set up; useful for low-securityenvironments, e.g. subscription-based online newspaper
6.3.2. DIGEST
6.3.2.1. variation to BASIC scheme; instead of transmitting password over networkdirectly, digest of password used instead, produced by taking a hash ofusername, password, uri, http method, and a randomly generated *nonce* value provided by server. Server computes digest as well, and compares with usersubmitted digest.
6.3.2.2. Advantage: transactions are somewhat more secure than with basicauthentication, since each digest is valid for only a single uri request and*nonce* value.
6.3.2.3. Disadvantage: server must still maintain a database of the originalpasswords
6.3.2.4. Disadvantage: digest authentication is not supported by very manybrowsers
6.3.3. FORM
6.3.3.1. the login page must include a form with a POST to the URL "j_security_check" with a username sent as j_username and a passwordj_password.
6.3.3.2. any timethe server receives a request for a protected resource, the server checks if theuser hasalready logged in, e.g. server might look for Principal object in HttpSessionobject. If Principal found, then roles are checked against security contraints;if Principal not authorized, then client redirected to <form-error-page>
6.3.3.3. Advantage: allows users to enter your site through a well-designed,descriptive and friendly login page
6.3.3.4. Disadvantage: similar to BASIC, password is transmitted in cleartext, unless SSL used
6.3.3.5. Disadvantage: similar to BASIC, no standard logout mechanism(calling session.invalidate() may work for FORM, but no guarantees), would needto close browser
6.3.3.6. Disadvantage: error page does not have access to any specialinformation reporting why access was denied or even which page it should pointthe user at to try again
6.3.3.7. Disadvantage: similar to BASIC, relies on server to authenticate,so only captures username and password, not custom fields e.g. PIN #.
6.3.4. CLIENT-CERT
6.3.4.1. BASIC, even with SSL encryption, does not ensure strong client authenticationsince anyone could have guessed or gotten hold of client's username andpassword
6.3.4.2. uponaccessing a protected resource, the server requests the client'scertificate; the client then sends its signed certificate (many browsers require the client userenter a password before they will send the certificate), and the server verifiesthe certificate. If browser has no certificate, or if it is not authorized, thenaccess is denied.
6.3.4.3. Advantage: the client will never see a login page, although thebrowser may prompt for a password to unlock their certificate before it is sent
6.3.4.4. Disadvantages: users must obtain and install signed certificates,servers must maintain a database of all accepted public keys, and servers mustsupport SSL 3.0 in the first place.

7. Designing and Developing Thread-Safe Servlets

7.1. Identify which attribute scopes are thread-safe:
7.1.1. local variables: yes, thread-safe
7.1.2. instancevariables: not thread-safe, since a single servlet instance may be handlingmultiple service requests at any given time
7.1.3. classvariables: not thread-safe, since multiple servlets and/or service requests maytry to access a class variable concurrently

7.1.4. requestattributes: yes, thread-safe, since the request object is a local variable
7.1.5. sessionattributes: not thread-safe, since sessions are scoped at the web applicationlevel, hence the same session object can be accessed concurrently by multiple servlets and their service requests
7.1.6. contextattributes: not thread-safe, since the same context object can be accessedconcurrently by multiple servlets and their service requests

7.2. Identify correct statements about differences between multi-threaded and single-threaded servlet models
7.2.1. multi-thread model servlet:
7.2.1.1. one servlet instance per registered name
7.2.1.2. for eachservlet request, the server spawns a separate thread which executes theservlet's service() method
7.2.1.3. mustsynchronize access to instance variables
7.2.2. single-thread model servlet:
7.2.2.1. has a pool of servlet instances per registered name (depending on the serverimplementation, the pool size may be configurable or not, and may be as littleas one.)
7.2.2.2. guaranteed by server "that no two threads will executeconcurrently in the servlet's service method"
7.2.2.3. considered thread-safe and isn't required to synchronizeaccess to instance variables
7.2.2.4. does notprevent synchronization problems that result from servlets accessing shared resources such as static variables orobjects outside the scope of the servlet (e.g. ServletContext, HttpSession)
7.2.2.5. servermight end up creating more instances than the system can handle, e.g. eachinstancemight have its own db connection, hence in total there may be more dbconnections than the db server can handle.

7.3. Identify the interface used to declare that a servlet must use the single threadmodel
7.3.1. interface javax.servlet.SingleThreadModel { // this is an empty"tag" interface }

Section 8: The JSP Model

8.1 Valid Opening and Closing Tags for different tag types.

a) Directive - <%@ Directive {attribute="value"}* %>
b) Declaration - <%! JavaDeclaration %>
c) Scriptlet - <% JavaStatements %>
d) Expression - <%= JavaExpression %>

8.2 Purpose of the different tag types

a) Directive – controls translation and compilation phase.
b) Declaration – declare a java variable or method. No output produced. Class level declarations.
c) Scriptlet – valid java statements. No output produced unless out is used
d) Expression – evaluation of expression to a String and then included in the output.

8.3 XML Version of Tag Types

a) <jsp:directive.page …/>
b) <jsp:declaration> … </jsp:declaration>
c) <jsp:scriptlet> … </jsp:scriptlet>
d) <jsp:expression> … </jsp:expression>

8.4 Specified Page Directive examples

a) <%@ page import="java.util.Date" %>
b) <%@ session="true"
c) <%@ errorPage="errorPage.jsp" %>
d) <%@ page isErrorPage="true" %>

8.5 sequence in order as per requirements document.

8.6 Implicit Objects and their purpose

1) request – the request
2) response – the response
3) out – object for writing to the output stream.
4) session – created as long as <% page sessio="false" %> is not used.
5) config – Specific to a servlet instance. Same as Servlet getServletConfig()
6) application – ServletContext – Available to all Servlets (application wide).
7) Provides access to resources of the servlet engine (resources, attributes, init context parms, request dispatcher, server info, URL & MIME resources).
8) page – this instance of the page (equivalent servlet instance 'this').
9) pageContext – PageContext – Environment for this page. Used by Servlet engine to manage such features as error pages and parms for included/forwarded pages.
10) exception – Throwable – created only if <%@ page isErrorPage="true" %> this is an error page that has been forwarded to from a previous page in error.

8.7 Examples of conditional and iterative statements.

a) Conditional Statement

```
<% if (event.equals("ENTER_RECORD")) {
%>
 <jsp:include page="enterRecord.jsp" flush=true"/>
<%
  } else if (event.equals ("NEW_RECORD")) {
%>
 <jsp:include page="newRecord.jsp" flush="true/>
<%
} else {
 %>
 jsp:include page="default.jsp" flush = true"/>
<% } %>
```

b) Iterative Statement

```
<% Hashtable h = (Hashtable) session.getAttribute("charges");
       if (h != null)
      {
%>
 <ul>
<%
Enumeration charges = h.keys();
 While (charges.hasMoreElements())
            {
  String proj = (String) charges.nextElement();
  Charge ch = (Charge) h.get(proj);
%>
 <li>
name = <%= ch.getName() %>
, project = <%= proj %>
, hours = <%= ch.getHours() %>
, date = <%= ch.getDate() %>
<%
 }
%>
 </ul>
<%
     }
%>
```

Section 9: Designing and Developing Reusable Web Components

9.1 Given a description of required functionality, identify the JSP directive or standard tag in the correct format with the correct attributes required to specify the inclusion of a Web component into the JSP page

• Include Directive - Translation-time
<%@ include file=... %>
Content is parsed by JSP container.

• Include Action - Request-time
<jsp:include page=… />
Content is not parsed; it is included in place.

Custom Tags
How about scriplets, requestdispatcher includes?

Section 10: Designing and Developing JSPs Using JavaBeans

10.1
a) <jsp:useBean id="connection", class="com.myco.myapp.Connection", scope="page" />
b) <jsp:getProperty name="beanName" property="propName" />
Also, this usebean tag includes one time initialization in the body
<jsp:useBean id="beanName" scope="request" class="Popovits.name">
  <jsp:setProperty name="id" property="givenName" value="Michele" />
</jsp:useBean>
c) see a)
d) see a) & b)
e) <% String timeout = connection.getTimeout; %>
f) <jsp:getProperty name="connection", property="timeout"/>
g) <jsp:setProperty name="connection", property="timeout", value="33"/>

10.2 Servlet Equivalent

request - HTTPServletRequest
session – HTTPServletRequest.getSession() : HTTPSession
application – GenericServlet.getServletContext() : ServletContext or
GenericServlet.getServletConfig().getServletContext()

10.3 Techniques for accessing a declared java bean
A declared JavaBean can also be accessed using:
- ✓ Scriptlets
- ✓ Expressions
- ✓ Custom Tags

Section 11: Designing and Developing JSPs Using Custom Tags

11.1 web.xml tag library declaration

```
<web-app>
  <taglib>
    <taglib-uri>
      http://www.jspinsider.com/jspkit/javascript
    </taglib-uri>
    <taglib-location>
      /WEB-INF/JavaScriptExampleTag.tld
    </traglib-location>
  </taglib>
</web-app>
```

11.2 <%@ taglib uri=http://jakarta.apache.org/taglibs/datetime-1.0 prefix="dt" %>
- uri is defined in web.xml, prefix is used in the jsp to reference the tag.

11.3
a) <msgBean:message/>
b) <msgBean:message attrName="value" />
c) <msgBean:message> <h1>This is the title</h1> </msgBean>
d) <msgBean:message> <helloBean:sayHello/> </msgBean>

Section 12: Designing and Developing A Custom Tag Library

12.1 TLD Elements
a) <name>
b) <tag-class>
c) <body-content>
d) <attribute> <name>username</name> <rtexprvalue>true</rtexprvalue>
<required>false</required></attribute>

12.2 TLD Attribute Elements
a) <name>
b) <required>
c) <rtexprvalue> run time expression value

12.3 Valid bodycontact TLD values
a) empty
b) JSP
c) tagdependent

12.4
- ✓ doStart – Process the start tag for this instance
- ✓ doAfterBody – Process body (re)evaluation – repeat for iteration tag.
- ✓ doEndTag – Process the end tag for this instance. Used to clean up resources and add any closing tags to the output as necessary

12.5 Valid Return Values

a) doStartTag - Tag.EVAL_BODY_INCLUDE, BodyTag.EVAL_BODY_BUFFERED, SKIP_BODY
b) doAfterBody - EVAL_BODY_AGAIN, SKIP_BODY
c) doEndTag - EVAL_PAGE, SKIP_PAGE
d) PageContext.getOut - javax.servlet.jsp.JspWriter

	doStart	doAfterBody	doEndTag
EVAL_BODY_INCLUDE	All		
EVAL_BODY_BUFFERED	Body tag only		
EVAL_BODY_AGAIN		Iteration	
SKIP_BODY	All	All	
EVAL_BODY_TAG	*deprecated*		
EVAL_PAGE			All
SKIP_PAGE			All

12.6
EVAL_BODY_AGAIN: Request the reevaluation of some body. Returned from doAfterBody. For compatibility with JSP 1.1, the value is carefully selected to be the same as the, now deprecated, BodyTag.EVAL_BODY_TAG,
EVAL_BODY_BUFFERED: Request the creation of new buffer, a BodyContent on which to evaluate the body of this tag. Returned from doStartTag when it implements BodyTag. This is an illegal return value for doStartTag when the class does not implement BodyTag.
EVAL_BODY_INCLUDE: Evaluate body into existing out stream. Valid return value for doStartTag.
SKIP_BODY: Skip body evaluation. Valid return value for doStartTag and doAfterBody.
EVAL_PAGE: Continue evaluating the page. Valid return value for doEndTag().
SKIP_PAGE: Skip the rest of the page. Valid return value for doEndTag.

12.7

a) pageContext.getSession();
b) pageContext.getRequest.getAttribute("attrName");

12.8
Tag Tag.getParent()
Tag TagSupport.getParent()
Tag TagSupport.findAncestorWithClass(Tag from, java.lang.Class class)