Java SE 11 Revision

1Z0-819 Exam

Chapter 2: Java Building Blocks

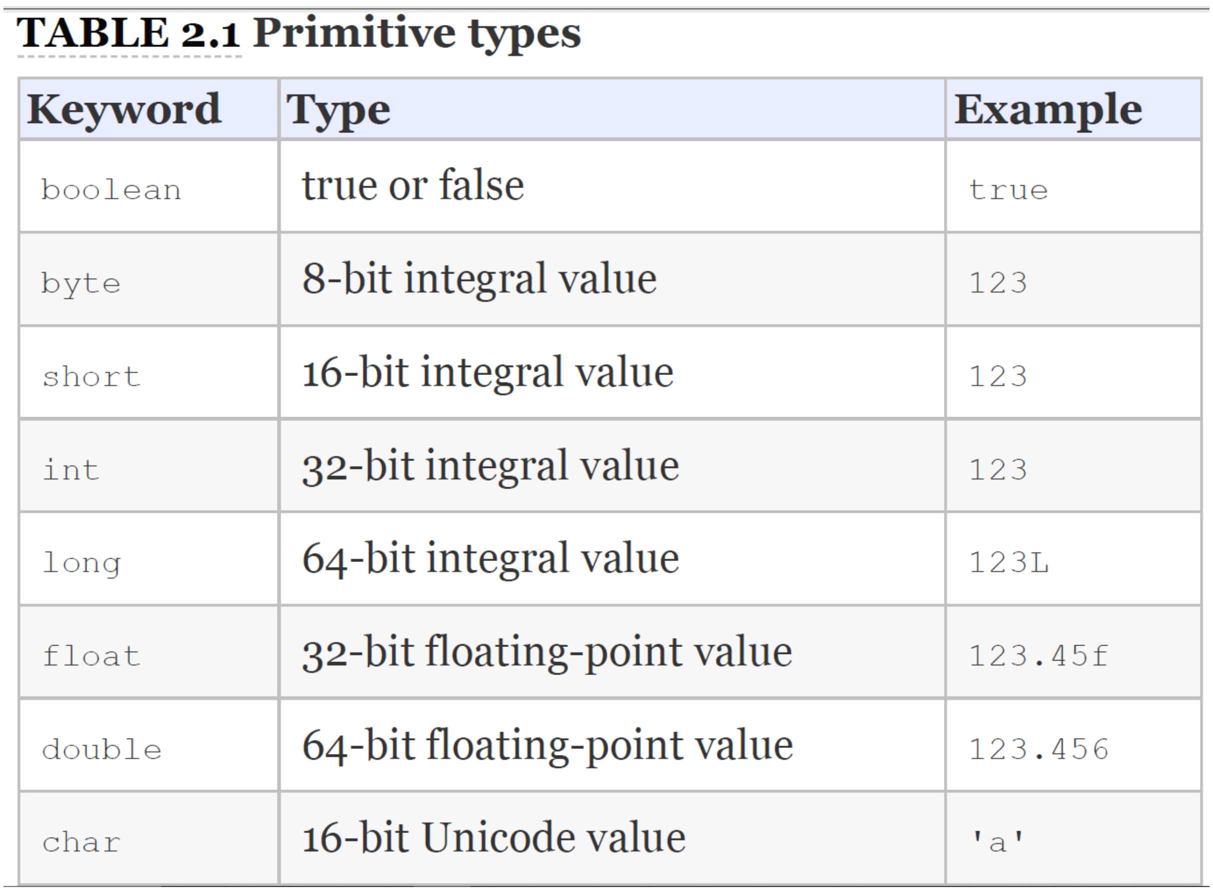


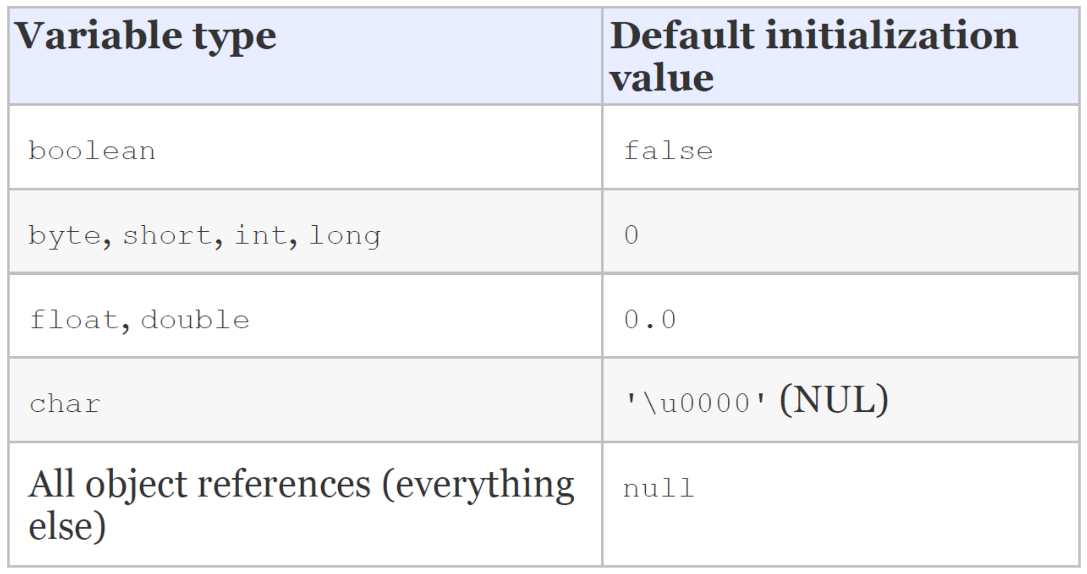
TABLE 2.1 Primitive types

Keyword	Type	Example
`boolean`	true or false	`true`
`byte`	8-bit integral value	`123`
`short`	16-bit integral value	`123`
`int`	32-bit integral value	`123`
`long`	64-bit integral value	`123L`
`float`	32-bit floating-point value	`123.45f`
`double`	64-bit floating-point value	`123.456`
`char`	16-bit Unicode value	`'a'`

Prefix

- Octal (digits 0–7), which uses the number 0 as a prefix—for example, 017
- Hexadecimal (digits 0–9 and letters A–F/a–f), which uses 0x or 0X as a prefix—for example, 0xFF, 0xff, 0XFf.
- Binary (digits 0–1), which uses the number 0 followed by b or B as a prefix—for example, 0b10, 0B10

TABLE 2.3 Default initialization values by type



Variable type	Default initialization value
`boolean`	`false`
`byte`, `short`, `int`, `long`	`0`
`float`, `double`	`0.0`
`char`	`'\u0000'` (NUL)
All object references (everything else)	`null`

Review of var Rules

We complete this section by summarizing all of the various rules for using var in your code. Here's a quick review of the var rules:

1. A var is used as a local variable in a constructor, method, or initializer block.
2. A var cannot be used in constructor parameters, method parameters, instance variables, or class variables.
3. A var is always initialized on the same line (or statement) where it is declared.
4. The value of a var can change, but the type cannot.
5. A var cannot be initialized with a null value without a type.
6. A var is not permitted in a multiple-variable declaration.
7. A var is a reserved type name but not a reserved word, meaning it can be used as an identifier except as a class, interface, or enum name.

REVIEWING SCOPE

Local variables: In scope from declaration to end of block.

Instance variables: In scope from declaration until object eligible for garbage collection.

Class variables: In scope from declaration until program ends Not sure what garbage collection is? Relax, that's our next and final section for this chapter.

Garbage Collection

An object is no longer reachable when one of two situations occurs:

1. The object no longer has any references pointing to it.
2. All references to the object have gone out of scope.

If any of the above condition is satisfied by a object **it is eligible for garbage collection.**

Summary

In this chapter, we described the building blocks of Java—most important, what a Java object is, how it is referenced and used, and how it is destroyed. This chapter lays the foundation for many topics that we will revisit throughout this book.

For example, we will go into a lot more detail on primitive types and how to use them in Chapter 3. Creating methods will be covered in Chapter 7. And in Chapter 8, we will discuss numerous rules for creating and managing objects. In other words, learn the basics, but don't worry if you didn't follow everything in this chapter. We will go a lot deeper into many of these topics in the rest of the book.

To begin with, constructors create Java objects. A constructor is a method matching the class name and omitting the return type. When an object is instantiated, fields and blocks of code are initialized first. Then the constructor is run.

Next, primitive types are the basic building blocks of Java types. They are assembled into reference types. Reference types can have methods and be assigned to null. Numeric literals are allowed to contain underscores (_) as long as they do not start or end the literal and are not next to a decimal point (.).

Declaring a variable involves stating the data type and giving the variable a name. Variables that represent fields in a class are automatically initialized to their corresponding 0, null, or false values during object instantiation. Local variables must be specifically initialized before they can be used. Identifiers may contain letters, numbers, $, or _. Identifiers may not begin with numbers. Local variables may use the var keyword instead of the actual type. When using var, the type is set once at compile time and does not change.

Moving on, scope refers to that portion of code where a variable can be accessed. There are three kinds of variables in Java, depending on their scope: instance variables, class variables, and local variables. Instance variables are the non-static fields of your class. Class variables are the static fields within a class. Local variables are declared within a constructor, method, or initializer block.

Finally, garbage collection is responsible for removing objects from memory when they can never be used again. An object becomes eligible for garbage collection when there are no more references to it or its references have all gone out of scope.

Exam Essentials

Be able to recognize a constructor. A constructor has the same name as the class. It looks like a method without a return type.

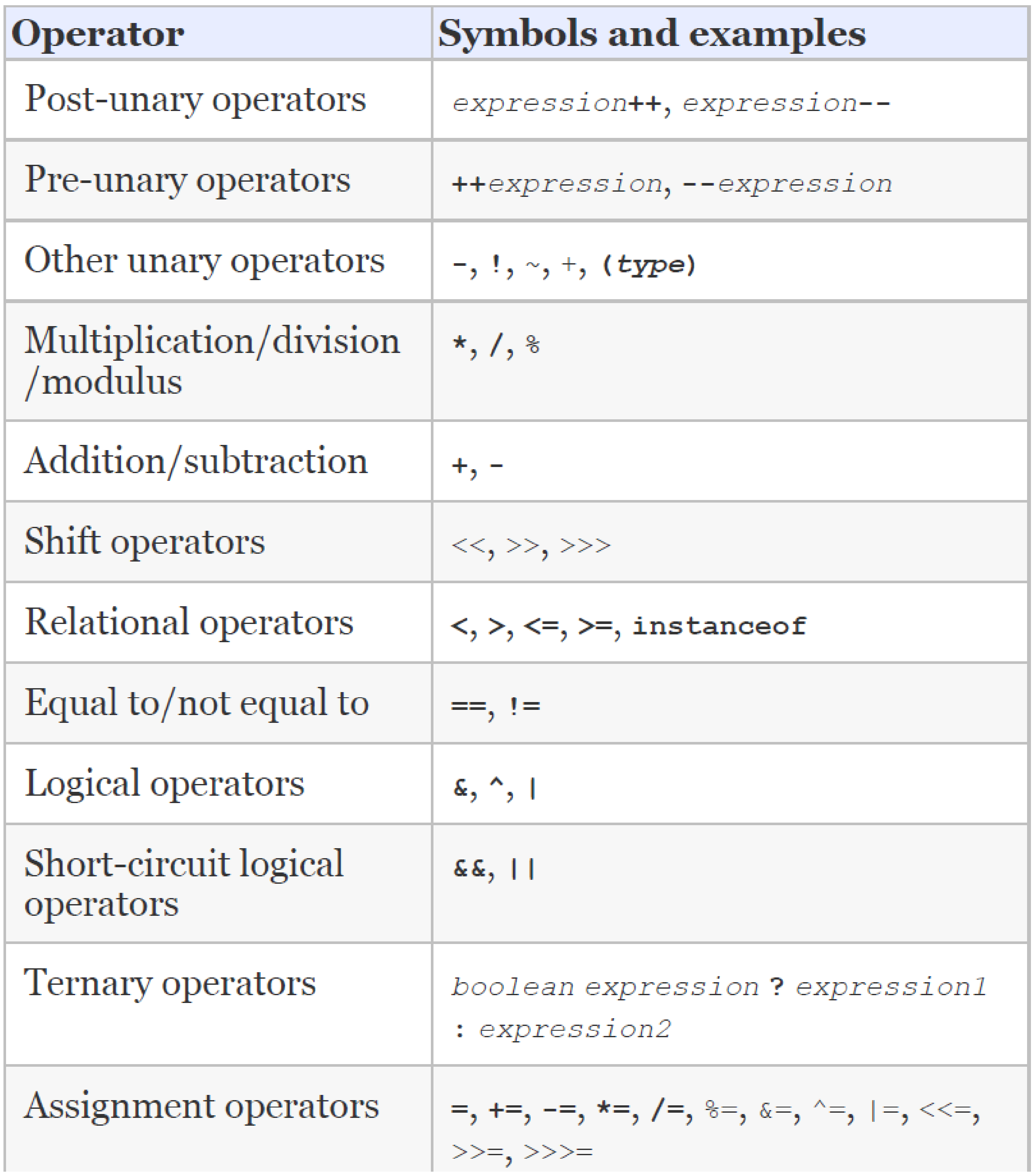
Be able to identify legal and illegal declarations and initialization. Multiple variables can be declared and initialized in the same statement when they share a type. Local variables require an explicit initialization; others use the default value for that type. Identifiers may contain letters, numbers, `$`, or _, although they may not begin with numbers. Also, you cannot define an identifier that is just a single underscore character _. Numeric literals may contain underscores between two digits, such as `1_000`, but not in other places, such as `_100_.0_`. Numeric literals can begin with `1–9`, `0`, `0x`, `0X`, `0b`, and `0B`, with the latter four indicating a change of numeric base.

Be able to use `var` correctly. A `var` is used for a local variable inside a constructor, a method, or an initializer block. It cannot be used for constructor parameters, method parameters, instance variables, or class variables. A `var` is initialized on the same line where it is declared, and while it can change value, it cannot change type. A `var` cannot be initialized with a `null` value without a type, nor can it be used in multiple variable declarations. Finally, `var` is not a reserved word in Java and can be used as a variable name.

Be able to determine where variables go into and out of scope. All variables go into scope when they are declared. Local variables go out of scope when the block they are declared in ends. Instance variables go out of scope when the object is eligible for garbage collection. Class variables remain in scope as long as the program is running.

Know how to identify when an object is eligible for garbage collection. Draw a diagram to keep track of references and objects as you trace the code. When no arrows point to a box (object), it is eligible for garbage collection.

Chapter 3: Operators



Operator	Symbols and examples
Post-unary operators	*expression*++, *expression*--
Pre-unary operators	++*expression*, --*expression*
Other unary operators	-, !, ~, +, (*type*)
Multiplication/division /modulus	*, /, %
Addition/subtraction	+, -
Shift operators	<<, >>, >>>
Relational operators	<, >, <=, >=, instanceof
Equal to/not equal to	==, !=
Logical operators	&, ^, |
Short-circuit logical operators	&&, ||
Ternary operators	*boolean expression* ? *expression1* : *expression2*
Assignment operators	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=

Note:

Unlike some other programming languages, in Java, 1 and true are not related in any way, just as 0 and false are not related.

TABLE 3.2 Unary operators

Operator	Description
`!`	Inverts a boolean's logical value
`+`	Indicates a number is positive, although numbers are assumed to be positive in Java unless accompanied by a negative unary operator
`-`	Indicates a literal number is negative or negates an expression
`++`	Increments a value by 1
`--`	Decrements a value by 1
`(type)`	Casts a value to a specific type.

TABLE 3.3 Binary arithmetic operators

Operator	Description
`+`	Adds two numeric values
`-`	Subtracts two numeric values
`*`	Multiplies two numeric values
`/`	Divides one numeric value by another
`%`	Modulus operator returns the remainder after division of one numeric value by another

Numeric Promotion Rules

1. If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.

2. If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value's data type.

3. Smaller data types, namely, `byte`, `short`, and `char`, are first promoted to `int` any time they're used with a Java binary arithmetic operator, even if neither of the operands is `int`.

4. After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.

ASSIGNMENT OPERATOR RETURN VALUE

the result of an assignment is an expression in and of itself, equal to the value of the assignment.

eg:

```
long wolf = 5;
long coyote = (wolf=3);
System.out.println(wolf); // 3
System.out.println(coyote); // 3
```

TABLE 3.6 Equality operators

Operator	Apply to primitives	Apply to objects
`==`	Returns `true` if the two values represent the same value	Returns `true` if the two values reference the same object
`!=`	Returns `true` if the two values represent different values	Returns `true` if the two values do not reference the same object

TABLE 3.7 Relational operators

Operator	Description
`<`	Returns `true` if the value on the left is strictly less than the value on the right
`<=`	Returns `true` if the value on the left is less than or equal to the value on the right
`>`	Returns `true` if the value on the left is strictly greater than the value on the right
`>=`	Returns `true` if the value on the left is greater than or equal to the value on the right
a `instanceof` *b*	Returns `true` if the reference that `a` points to is an instance of a class, subclass, or class that implements a particular interface, as named in `b`

Note: The first four relational operators in Table 3.7 apply only to numeric values. If the two numeric operands are not of the same data type, the smaller one is promoted as previously discussed.

```
System.out.print(null instanceof AnyThingHere); // false
System.out.print(null instanceof null); // DOES NOT COMPILE
```

The logical operators

(`&`), (`|`), and (`^`), may be applied to both numeric and `boolean` data types; they are listed in Table 3.8. When they're applied to `boolean` data types, they're referred to as *logical operators*. Alternatively, when they're applied to numeric data types, they're referred to as *bitwise operators*,

TABLE 3.9 **Short-circuit operators**

Operator	Description
`&&`	Short-circuit AND is `true` only if both values are `true`. If the left side is `false`, then the right side will not be evaluated.
`||`	Short-circuit OR is `true` if at least one of the values is `true`. If the left side is `true`, then the right side will not be evaluated.

Be wary of short-circuit behavior on the exam, as questions are known to alter a variable on the right side of the expression that may never be reached. This is referred to as an *unperformed side effect*.

NOTE: Ternary operator and Short-circuit operators can lead to unperformed side effects.

Exam Essentials

Be able to write code that uses Java operators. This chapter covered a wide variety of operator symbols. Go back and review them several times so that you are familiar with them throughout the rest of the book.

Be able to recognize which operators are associated with which data types. Some operators may be applied only to numeric primitives, some only to `boolean` values, and some only to objects. It is important that you notice when an operator and operand(s) are mismatched, as this issue is likely to come up in a couple of exam questions.

Understand when casting is required or numeric promotion occurs. Whenever you mix operands of two different data types, the compiler needs to decide how to handle the resulting data type. When you're converting from a smaller to a larger data type, numeric promotion is automatically applied. When you're converting from a larger to a smaller data type, casting is required.

Understand Java operator precedence. Most Java operators you'll work with are binary, but the number of expressions is often greater than two. Therefore, you must understand the order in which Java will evaluate each operator symbol.

Be able to write code that uses parentheses to override operator precedence. You can use parentheses in your code to manually change the order of precedence.

Chapter 4: Making Decisions

The `switch` value and data type should be compatible with the `case` statements, and the values for the `case` statements must evaluate to compile-time constants. Finally, at runtime a `switch` statement branches to the first matching `case`, or `default` if there is no match, or exits entirely if there is no match and no `default` branch. The process then continues into any proceeding `case` or `default` statements until a `break` or `return` statement is reached.

The following is a list of all data types supported by `switch` statements:

1. `int` and `Integer`
2. `byte` and `Byte`
3. `short` and `Short`
4. `char` and `Character`
5. `String`
6. `enum` values
7. `var` (if the type resolves to one of the preceding types)

Acceptable Case Values

we can use only literals, `enum` constants, or `final` constant variables of the same data type in case value.

```
final String str = “akash”;
final int aaa = 10;
switch(“akash”){
        case str: statement; break;
        case aaa: statement; // DOES NOT COMPILE
}
```

final variable passed to a method and then use that in switch will also not work.

for enum, the case value must be in the enum and must be used in case like CONSTANT and not like ENUM_NAME.CONSTANT.

switch **do not allow** continue statement.

For Loop

```
for(int y = 0, int z = 4; x < 5; x++) { // COMPILE
for(long y = 0, int z = 4; x < 5; x++) { // DOES NOT COMPILE
```

The variables in the initialization block must all be of the same type.

For-each Loop

The for-each loop declaration is composed of an initialization section and an object to be iterated over. The right side of the for-each loop must be one of the following:

1. A built-in Java array
2. An object whose type implements `java.lang.Iterable` (like List, Set, etc.)

Note:

1. left side and right side type should be same.
2. `Map, String` and `StringBuilder`, do not implement `Iterable` and cannot be used as the right side of a for-each statement.
3. A `var` may also be used for the variable type declaration

UNREACHABLE CODE

One facet of `break`, `continue`, and `return` that you should be aware of is that any code placed immediately after them in the same block is considered unreachable and will not compile.

```
if(200>100) {
     break;
     checkDate++; // DOES NOT COMPILE
}
```

```
switch(5) {
case 1: return 1; hour++; // DOES NOT COMPILE
}
```

<u>Chapter 5: Core Java APIs</u>

String methods:

```
int length()
```

```
char charAt(int index)
```

```
int indexOf(int ch)
int indexOf(int ch, int fromIndex)
int indexOf(String str)
int indexOf(String str, int fromIndex)
```

```
String substring(int beginIndex)
String substring(int beginIndex, int endIndex)
```

```
String toLowerCase()
String toUpperCase()
```

```
boolean equals(Object obj)
boolean equalsIgnoreCase(String str)
```

```
boolean startsWith(String prefix)
boolean endsWith(String suffix)
```

```
String replace(char oldChar, char newChar)
String replace(CharSequence target, CharSequence replacement)
```

```
boolean contains(CharSequence charSeq)
```

```
String strip()
String stripLeading()
String stripTrailing()
String trim()
```

The `intern()` method returns the value from the string pool if it is there. Otherwise, it adds the value to the string pool.

```
String intern()
```

CREATING A *STRINGBUILDER*

There are three ways to construct a `StringBuilder`:

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("animal");
StringBuilder sb3 = new StringBuilder(10); // 10 is initial
capacity
```

STRINGBUILDER METHODS

charAt(), indexOf(), length(), and substring() same as String

```
StringBuilder append(String str)
```

```
StringBuilder insert(int offset, String str)
```

The `delete()` method is more flexible than some others when it comes to array indexes. If you specify a second parameter that is past the end of the `StringBuilder`, Java will just assume you meant the end.

```
StringBuilder delete(int startIndex, int endIndex)
StringBuilder deleteCharAt(int index)
```

```
StringBuilder replace(int start, int end, String newStr)
```

Internally replace calls delete then insert
like,

```
StringBuilder delete(int start, int end)
and then
StringBuilder insert(int start,String newStr)

StringBuilder reverse()
```

```
String toString()
```

THE *STRING* POOL

```
String name = "Hello World";
String name2 = new String("Hello World").intern();
System.out.println(name == name2); // true
```

```
String first = "rat" + 1;
String second = "r" + "a" + "t" + "1";
System.out.println(first == second); // true
```

NOTE: if the string is a compile time constant the it is lookedup in the pool and if there is a same string object it points to that.

Arrays

down casting an array.

```
3: String[] strings = { "stringValue" };
4: Object[] objects = strings;
5: String[] againStrings = (String[]) objects;
```

COMPARING TWO ARRAYS

compare()

- If both arrays are the same length and have the same values in each spot in the same order, return zero.
- If all the elements are the same but the second array has extra elements at the end, return a negative number.
- If all the elements are the same but the first array has extra elements at the end, return a positive number.
- If the first element that differs is smaller in the first array, return a negative number.
- If the first element that differs is larger in the first array, return a positive number.

mismatch()

If the arrays are equal, `mismatch()` returns `-1`. Otherwise, it returns the first index where they differ.

TABLE 5.3 Equality vs. comparison vs. mismatch

Method	When arrays are the same	When arrays are different
`equals()`	`true`	`false`
`compare()`	`0`	Positive or negative number
`mismatch()`	`-1`	Zero or positive index

Multi-dimensional Array

```
int[] a, b;  // two int array

int a[], b; // a is int array and b is int variable

int[] a[], b; // a is 2D and b is 1D
```

ARRAYLIST

CREATING AN *ARRAYLIST*

As with `StringBuilder`, there are three ways to create an `ArrayList`:

```
ArrayList list1 = new ArrayList();
ArrayList list2 = new ArrayList(10);
ArrayList list3 = new ArrayList(list2);
```

Also since java 5,

```
ArrayList<String> list4 = new ArrayList<String>();
```

```
ArrayList<String> list5 = new ArrayList<>();
```

ARRAYLIST Methods

1. `boolean add(E element)`
   `void add(int index, E element)`

2. `boolean remove(Object object)`
   `removedElement remove(int index)`

3. `E set(int index, E newElement)`

4. `boolean isEmpty()`

5. `int size()`

6. `void clear()`

7. `boolean contains(Object object)`

8. `boolean equals(Object object)`

Wrapper classes

1. From string to primitive

```
Boolean.parseBoolean("true")
```

2. From string to wrapper class

```
Boolean.valueOf("TRUE")
```

Character class does not have above methods

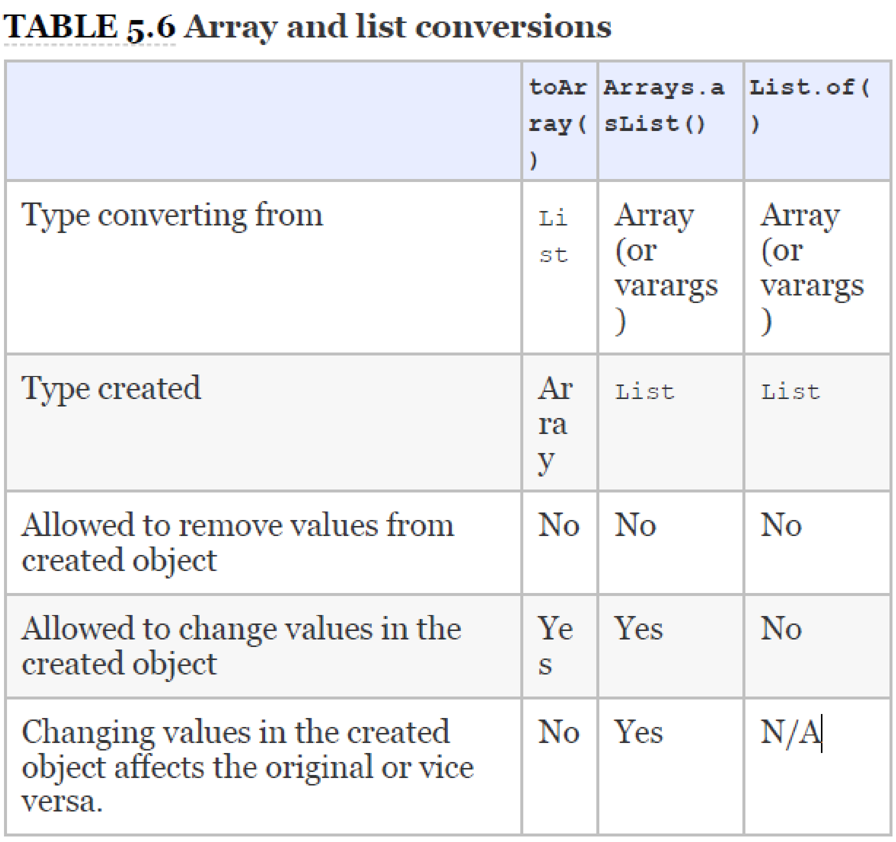


TABLE 5.6 Array and list conversions

	`toArray()`	`Arrays.asList()`	`List.of()`
Type converting from	`List`	Array (or varargs)	Array (or varargs)
Type created	Array	`List`	`List`
Allowed to remove values from created object	No	No	No
Allowed to change values in the created object	Yes	Yes	No
Changing values in the created object affects the original or vice versa.	No	Yes	N/A

Math class methods

java.lang.Math

```
1. double min(double a, double b)
   float min(float a, float b)
   int min(int a, int b)
   long min(long a, long b)



2. long round(double num)
   int round(float num)


3. double pow(double number, double exponent)


4. double num = Math.random(); // [0,1)
```

Summary

In this chapter, you learned that `String`s are immutable sequences of characters. The `new` operator is optional. The concatenation operator (`+`) creates a new `String` with the content of the first `String` followed by the content of the second `String`. If either operand involved in the + expression is a `String`, concatenation is used; otherwise, addition is used. `String` literals are stored in the string pool. The `String` class has many methods.

`StringBuilder`s are mutable sequences of characters. Most of the methods return a reference to the current object to allow method chaining. The `StringBuilder` class has many methods. Calling `==` on `String` objects will check whether they point to the same object in the pool. Calling `==` on `StringBuilder` references will check whether they are pointing to the same `StringBuilder` object. Calling `equals()` on `String` objects will check whether the sequence of characters is the same. Calling `equals()` on `StringBuilder` objects will check whether they are pointing to the same object rather than looking at the values inside.

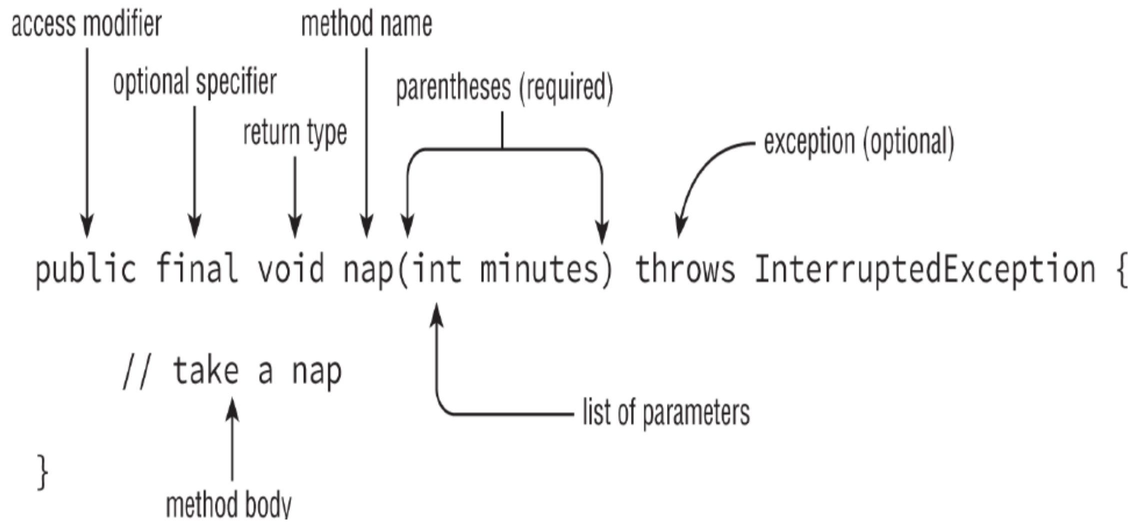
An array is a fixed-size area of memory on the heap that has space for primitives or pointers to objects. You specify the size when creating it—for example, `int[] a = new int[6];`. Indexes begin with 0, and elements are referred to using `a[0]`. The `Arrays.sort()` method sorts an array. `Arrays.binarySearch()` searches a sorted array and returns the index of a match. If no match is found, it negates the position where the element would need to be inserted and subtracts 1. `Arrays.compare()` and `Arrays .mismatch()` check whether two arrays are the equivalent. Methods that are passed varargs (...) can be used as if a normal array was passed in. In a multidimensional array, the second-level arrays and beyond can be different sizes.
An `ArrayList` can change size over its life. It can be stored in an `ArrayList` or `List` reference. Generics can specify the type that goes in the `ArrayList`. Although an `ArrayList` is not allowed to contain primitives, Java will autobox parameters passed in to the proper wrapper type. `Collections.sort()` sorts an `ArrayList`.

A `Set` is a collection with unique values. A `Map` consists of key/value pairs. The `Math` class provides many static methods to facilitate programming.

Chapter 6: Lambdas and Functional Interfaces

refer chapter 12

Chapter 7: Methods and Encapsulation



```
public final void nap(int minutes) throws InterruptedException {
    // take a nap
}
```

A varargs parameter must be the last element in a method's parameter list. This means you are allowed to have only one varargs parameter per method.

ACCESS MODIFIERS

Java offers four choices of access modifier:

private: The private modifier means the method can be called only from within the same class. | Only accessible within the same class.

Default (Package-Private) Access: With default access, the method can be called only from classes in the same package. This one is tricky because there is no keyword for default access. You simply omit the access modifier. | **private** plus other classes in the same package.

protected: The protected modifier means the method can be called only from classes in the same package or subclasses. | **Default access** plus child classes.

public: all classes in all packages have access. | **protected** plus classes in the other packages

OPTIONAL SPECIFIERS

Unlike with access modifiers, you can have multiple specifiers in the same method (although not all combinations are legal). When this happens, you can specify them in any order. And since these specifiers are optional, you are allowed to not have any of them at all. This means you can have zero or more specifiers in a method declaration.

static: The static modifier is used for class methods.

abstract: The abstract modifier is used when a method body is not provided.

final: The final modifier is used when a method is not allowed to be overridden by a subclass.

synchronized: The synchronized modifier is used with multithreaded code.

The **protected rules** apply under **two** scenarios:

1. A member is used without referring to a reference variable. In this case, we are taking advantage of inheritance and protected access is allowed.
2. A member is used through a reference variable. In this case, the rules for the reference type of the variable are what matter. If

reference type of the reference variable is a subclass, protected access is allowed (provided it is calling inside the subclass). This works for references to the same class (provided it is calling inside the same package) or a subclass (provided it is calling inside the subclass).
(if not understanding read pg. 486)

STATIC VS. INSTANCE

Static Belongs to Class and object.

Non-Static/Instance belong to only object

____ method/variable can call _____ Method/variable directly.	Non-Static	Static
Non-Static	YES	YES
Static	NO*	YES

*Static method/variable can call Non-static method/variable using reference of that class

When accessing a static variable/method using the reference of the object of the class, JVM don't care if the reference if pointing to object or not (pointing to null). It checks the type of the reference variable and call the static variable/method using the class name only.

example:

```
public class Koala {
     public static int count = 0; // static variable
     public static void main(String[] args) { // static method
          System.out.println(count);
     }
}

Koala k = new Koala();
System.out.println(k.count); // k is a Koala
k = null;
System.out.println(k.count); // k is still a Koala
```

output:

0

0

Overloading Methods

rules

1. method name **must** be same.
2. Parameter list **must** be changed (number of parameter, type of parameter, order of parameter).
3. return type **can be** changed.
4. access modifier **can be** changed.
5. exception list **can be** changed.

Note:

overloading generic method type

```
public void walk(List<String> strings) {}
public void walk(List<Integer> integers) {} // DOES NOT
COMPILE
```

Java treats it as List in both methods.

```
public void fly(int[] lengths) {}
public void fly(int... lengths) {} // DOES NOT COMPILE
```

Java treats **varargs as if they were an array**. This means that the method signature is the same for both methods. Since we are not allowed to overload methods with the same parameter list, this code doesn’t compile. Even though the code doesn’t look the same, it compiles to the same parameter list.

NOTE: From within a method, an array or varargs parameter is treated the same. However, there is a difference from the caller’s point of view. A varargs parameter can receive either an array or individual values.

```
public void walk(List<String> strings) {}
public void walk(List<Integer> integers) {} // DOES NOT
COMPILE
```

Java has a concept called *type erasure* where generics are used only at compile time. That means the compiled code looks like this:

```
public void walk(List strings) {}
public void walk(List integers) {} // DOES NOT COMPILE
```

Unlike the previous example, this code is just fine:

```
public static void walk(int[] ints) {}
public static void walk(Integer[] integers) {}
```

Arrays have been around since the beginning of Java. They specify their actual types and don't participate in type erasure.

TABLE 7.4 The order that Java uses to choose the right overloaded method

Rule	**Example of what will be chosen for `glide(1,2)`**
Exact match by type	`String glide(int i, int j)`
Larger primitive type	`String glide(long i, long j)`
Autoboxed type	`String glide(Integer i, Integer j)`
Varargs	`String glide(int... nums)`

while trying to find a match, java will do only one conversion.

eg: int -> Integer or String -> Object

but will not, int -> Integer -> Long or int -> Integer -> Object

Summary

As you learned in this chapter, Java methods start with an access modifier of `public`, `private`, `protected`, or blank (default access). This is followed by an optional specifier such as `static`, `final`, or `abstract`. Next comes the return type, which is `void` or a Java type. The method name follows, using standard Java identifier rules. Zero or more parameters go in parentheses as the parameter list. Next come any optional exception types. Finally, zero or more statements go in braces to make up the method body.

Using the `private` keyword means the code is only available from within the same class. Default (package-private) access means the code is available only from within the same package. Using the `protected` keyword means the code is available from the same package or subclasses. Using the `public` keyword means the code is available from anywhere. Both `static` methods and `static` variables are shared by all instances of the class. When referenced from outside the class, they are called using the classname—for example, `StaticClass.method()`. Instance members are allowed to call `static` members, but `static` members are not allowed to call instance members. Static imports are used to import `static` members.

Java uses pass-by-value, which means that calls to methods create a copy of the parameters. Assigning new values to those parameters in the method doesn't affect the caller's variables. Calling methods on objects that are method parameters changes the state of those objects and is reflected in the caller.

Overloaded methods are methods with the same name but a different parameter list. Java calls the most specific method it can find. Exact matches are preferred, followed by wider primitives. After that comes autoboxing and finally varargs.

Encapsulation refers to preventing callers from changing the instance variables directly. This is done by making instance variables `private` and getters/setters `public` (getters/setters can be private, default or protected). Encapsulation means instance variable should be private getter setter can be anything.

Exam Essentials

Be able to identify correct and incorrect method declarations. A sample method declaration is `public static void method(String... args) throws Exception {}`.

Identify when a method or field is accessible. Recognize when a method or field is accessed when the access modifier (`private`, `protected`, `public`, or default access) does not allow it.

Recognize valid and invalid uses of static imports. Static imports import `static` members. They are written as `import static`, not *static import*. Make sure they are importing `static` methods or variables rather than class names.

State the output of code involving methods. Identify when to call `static` rather than instance methods based on whether the class name or object comes before the method. Recognize that instance methods can call `static` methods and that `static` methods need an instance of the object in order to call an instance method.

Recognize the correct overloaded method. Exact matches are used first, followed by wider primitives, followed by autoboxing, followed by varargs. Assigning new values to method parameters does not change the caller, but calling methods on them does.

Identify properly encapsulated classes. Instance variables in encapsulated classes are `private`. All code that retrieves the value or updates it uses methods. These methods are allowed to be `public`.

Chapter 8: Class Design

If you try to define a class that inherits from a final class, then the class will fail to compile.

In Java, all classes inherit from a single class: java.lang.Object, or Object for short. Furthermore, Object is the only class that doesn't have a parent class.

In Java, a top-level class is a class that is not defined inside another class. They can only have public or package-private access. An inner class is a class defined inside of another class and is the opposite of a top-level class. In addition to public and package-private access, inner classes can also have protected and private access. a Java file can have many top-level classes but at most one public top-level class.

this

The `this` reference refers to the current instance of the class and can be used to access any member of the class, including inherited members. It can be used in any instance method, constructor, and instance initializer block. It cannot be used when there is no implicit instance of the class, such as in a `static` method or `static` initializer block.

super

The `super` reference is similar to the `this` reference, except that it excludes any members found in the current class. In other words, the member must be accessible via inheritance.

Remember, while `this` includes current and inherited members, `super` only includes inherited members.

Constructors

Like method parameters, constructor parameters can be any valid class, array, or primitive type, including generics, but may not include `var`.

A class can have multiple constructors, so long as each constructor has a unique signature.

this() and super()
this() and super() must be called only inside the constructor and that also only at the first line.

final class/instance variable do not get default value so,
All final variables must be initialized exactly once by the time constructor finishes compiling. all other cases will not compile.

Although a `final` instance variable can be assigned a value only once, each constructor is considered independently in terms of assignment. meaning not initialized final variables must be initialized in all the constructor.

Class Initialization

1. If there is a superclass Y of X, then initialize class Y first.
2. Process all `static` variable declarations in the order they appear in the class.
3. Process all `static` initializers in the order they appear in the class.

Instance Initialization

1. If there is a superclass Y of X, then initialize the instance of Y first.
2. Process all instance variable declarations in the order they appear in the class.
3. Process all instance initializers in the order they appear in the class.
4. Initialize the constructor including any overloaded constructors referenced with `this()`.

REVIEWING CONSTRUCTOR RULES

1. The first statement of every constructor is a call to an overloaded constructor via `this()`, or a direct parent constructor via `super()`.
2. If the first statement of a constructor is not a call to `this()` or `super()`, then the compiler will insert a no-argument `super()` as the first statement of the constructor.

3. Calling `this()` and `super()` after the first statement of a constructor results in a compiler error.
4. If the parent class doesn't have a no-argument constructor, then every constructor in the child class must start with an explicit `this()` or `super()` constructor call.
5. If the parent class doesn't have a no-argument constructor and the child doesn't define any constructors, then the child class will not compile.
6. If a class only defines `private` constructors, then it cannot be extended by a top-level class.
7. All `final` instance variables must be assigned a value exactly once by the end of the constructor. Any `final` instance variables not assigned a value will be reported as a compiler error on the line the constructor is declared.

Method Overriding

1. The method in the child class must have the same signature as the method in the parent class.
2. The method in the child class must be at least as accessible as the method in the parent class.
3. The method in the child class may not declare a checked exception that is new or broader than the class of any exception declared in the parent class method.
4. If the method returns a value, it must be the same or a subtype of the method in the parent class, known as *covariant return types*. eg: `CharSequence > String therefore OK.`
5. The method defined in the child class must be marked as `static` if it is marked as `static` in a parent class.
6. You cannot override or hide a **final** method, if done it will not compile. i.e final method cannot be overridden and final static method cannot be hidden.

Note:
1. A valid override of a method with **generic return type** must have a return type that is covariant, with **matching generic type** parameters. here generic class List and ArrayList are covariant.
eg: List<String> > ArrayList<String> therefore, are ok

2. A valid override of a method with **generic arguments** must

have the same signature with the **same generic types**.
// not sure about below part
you can change generic class (must be covariant) but generic type must be same
eg: List<String> > ArrayList<String> therefore, are ok

3. Java permits you to redeclare a new method in the child class with the same or modified signature as the **private** method in the parent class. This method in the child class is a separate and independent method, unrelated to the parent version's method, so, none of the rules for overriding methods is invoked.

4. it is method hiding if the two methods are marked `static`, and method overriding if they are not marked `static`. If one is marked `static` and the other is not, the class will not compile.

5. Java doesn't allow variables to be overridden. Variables can be hidden, though. A *hidden variable* occurs when a child class defines a variable with the same name as an inherited variable defined in the parent class. This creates two distinct copies of the variable within an instance of the child class: one instance defined in the parent class and one defined in the child class.

6. In polymorphic references (super to sub) all the methods and variables are used from the super type i.e the reference type and not the object type, **BUT** only Overridden methods are used from the subtype i.e from the object type not the reference type.

Summary

This chapter took the basic class structures we've presented throughout the book and expanded them by introducing the notion of inheritance. Java classes follow a multilevel single inheritance pattern in which every class has exactly one direct parent class, with all classes eventually inheriting from `java.lang.Object`.

Inheriting a class gives you access to all of the `public` and `protected` members of the class. It also gives you access to package-private members of the class if the classes are in the same package. All instance methods, constructors, and instance initializers have access to two special reference variables: `this` and `super`. Both `this` and `super` provide access to all inherited members, with only `this` providing access to all members in the current class declaration.

Constructors are special methods that use the class name and do not have a return type. They are used to instantiate new objects. Declaring constructors requires following a number of important rules. If no constructor is provided, the compiler will automatically insert a default no-argument constructor in the class. The first line of every constructor is a call to an overloaded constructor, `this()`, or a parent constructor, `super()`; otherwise, the compiler will insert a call to `super()` as the first line of the constructor. In some cases, such as if the parent class does not define a no-argument constructor, this can lead to compilation errors. Pay close attention on the exam to any class that defines a constructor with arguments and doesn't define a no-argument constructor.

Classes are initialized in a predetermined order: superclass initialization; `static` variables and `static` initializers in the order that they appear; instance variables and instance initializers in the order they appear; and finally, the constructor. All `final` instance variables must be assigned a value exactly once. If by the time a constructor finishes, a `final` instance variable is not assigned a value, then the constructor will not compile.

We reviewed overloaded, overridden, hidden, and redeclared methods and showed how they differ, especially in terms of polymorphism. A method is overloaded if it has the same name but a different signature as another accessible method. A method is overridden if it has the same signature as an inherited method, with access modifiers, exceptions, and a return type that are compatible. A `static` method is hidden if it has the same signature as an inherited static method. Finally, a method is redeclared if it has the same name and possibly the same signature as an uninherited method.

We also introduced the notion of hiding variables, although we strongly discourage this in practice as it often leads to confusing, difficult-to-maintain code.

Finally, this chapter introduced the concept of polymorphism, central to the Java language, and showed how objects can be accessed in a variety of forms. Make sure you understand when casts are needed for accessing objects, and be able to spot the difference between compile-time and runtime cast problems.

Exam Essentials

Be able to write code that extends other classes. A Java class that extends another class inherits all of its `public` and `protected` methods and variables. If the class is in the same package, it also inherits all package-private members of the class. Classes that are marked `final` cannot be extended. Finally, all classes in Java extend `java.lang.Object` either directly or from a superclass.

Be able to distinguish and make use of *this*, *this()*, *super*, and *super()*. To access a current or inherited member of a class, the `this` reference can be used. To access an inherited member, the `super` reference can be used. The `super` reference is often used to reduce ambiguity, such as when a class reuses the name of an inherited method or variable. The calls to `this()` and `super()` are used to access constructors in the same class and parent class, respectively.

Evaluate code involving constructors. The first line of every constructor is a call to another constructor within the class using `this()` or a call to a constructor of the parent class using the `super()` call. The compiler will insert a call to `super()` if no constructor call is declared. If the parent class doesn't contain a no-argument constructor, an explicit call to the parent constructor must be provided. Be able to recognize when the default constructor is provided. Remember that the order of initialization is to initialize all classes in the class hierarchy, starting with the superclass. Then, the instances are initialized, again starting with the superclass. All `final` variables must be assigned a value exactly once by the time the constructor is finished.

Understand the rules for method overriding. Java allows methods to be overridden, or replaced, by a subclass if certain rules are followed: a method must have the same signature, be at least as accessible as the parent method, must not declare any new or broader exceptions, and must use covariant return types. The generic parameter types must exactly match in any of the generic method arguments or a generic return type. Methods marked `final` may not be overridden or hidden.

Understand the rules for hiding methods and variables. When a `static` method is overridden in a subclass, it is referred to as method hiding. Likewise, variable hiding is when an inherited variable name is reused in a subclass. In both situations, the original method or variable still exists and is accessible depending on where it is accessed and the reference type used. For method hiding, the use of `static` in the method declaration must be the same between the parent and child class. Finally, variable and method hiding should generally be avoided since it leads to confusing and difficult to follow code.

Recognize the difference between method overriding and method overloading. Both method overloading and overriding involve creating a new method with the same name as an existing method. When the method signature is the same, it is referred to as method overriding and must follow a specific set of override rules to compile. When the method signature is different, with the method taking different inputs, it is referred to as method overloading, and none of the override rules are required. Method overriding is important to polymorphism because it replaces all calls to the method, even those made in a superclass.

Understand polymorphism. An object may take on a variety of forms, referred to as polymorphism. The object is viewed as existing in memory in one concrete form but is accessible in many forms through reference variables. Changing the reference type of an object may grant access to new members, but the members always exist in memory.

Recognize valid reference casting. An instance can be automatically cast to a superclass or interface reference without an explicit cast. Alternatively, an explicit cast is required if the reference is being narrowed to a subclass of the object. The Java compiler doesn’t permit casting to unrelated class types. Be able to discern between compiler-time casting errors and those that will not occur until runtime and that throw a `ClassCastException`.

Chapter 9: Advanced Class Design

Abstract Class Definition Rules

1. Abstract classes cannot be instantiated.
2. All top-level types, including abstract classes, cannot be marked `protected` or `private`.
3. Abstract classes cannot be marked `final`.
4. Abstract classes may include zero or more abstract and non-abstract methods.
5. An abstract class that `extends` another abstract class inherits all of its abstract methods.
6. The first concrete class that `extends` an abstract class must provide an implementation for all of the inherited abstract methods.
7. Abstract class constructors follow the same rules for initialization as regular constructors, except they can be called only as part of the initialization of a subclass.

These rules for abstract methods apply regardless of whether the abstract method is defined in an abstract class or interface.

Abstract Method Definition Rules

1. Abstract methods can be defined only in abstract classes or interfaces.
2. Abstract methods cannot be declared `private` or `final` or `static`.
3. Abstract methods must not provide a method body/implementation in the abstract class in which they are declared.
4. Implementing an abstract method in a subclass follows the same rules for overriding a method, including covariant return types, exception declarations, etc.

Like the `final` modifier, the `abstract` modifier can be placed before or after the access modifier in class and method declarations

The `abstract` modifier cannot be placed after the `class` keyword in a class declaration, nor after the return type in a method declaration.

Even though abstract classes cannot be instantiated, they are still initialized through constructors by their subclasses. therefore, they need constructor and follow all rules of compiler enhancement.

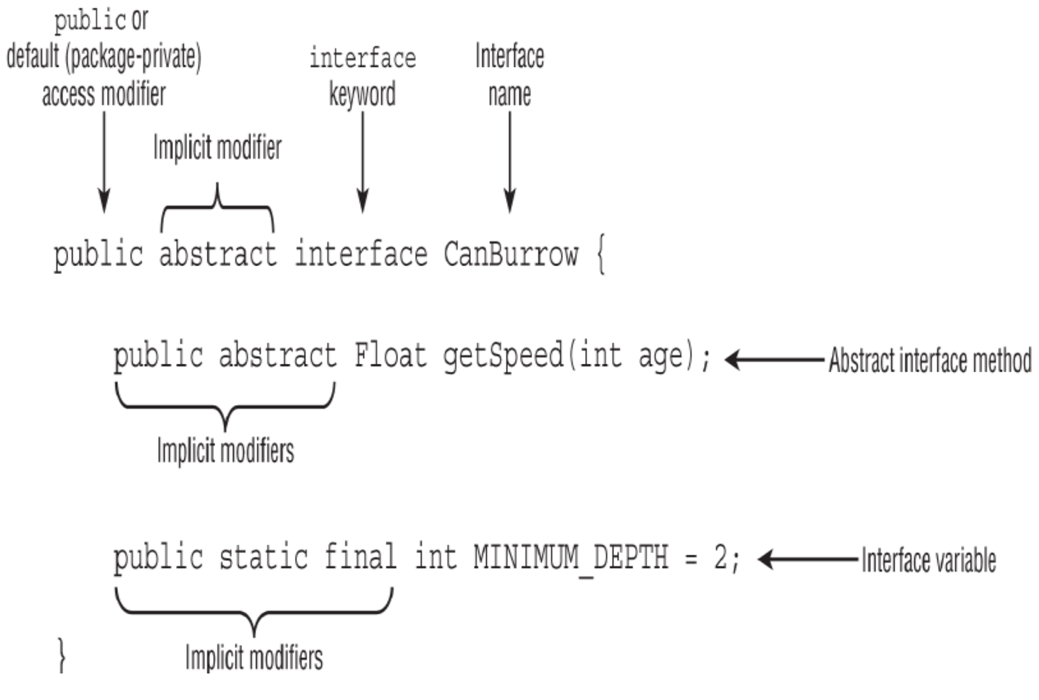
For the exam, remember that abstract classes are initialized with constructors in the same way as non-abstract classes. For example, if an abstract class does not provide a constructor, the compiler will automatically insert a default no-argument constructor.

The primary difference between a constructor in an abstract class and a nonabstract class is that a constructor in abstract class can be called only when it is being initialized by a nonabstract subclass. This makes sense, as abstract classes cannot be instantiated.

For the exam, remember that an abstract method declaration must end in a semicolon without any braces.

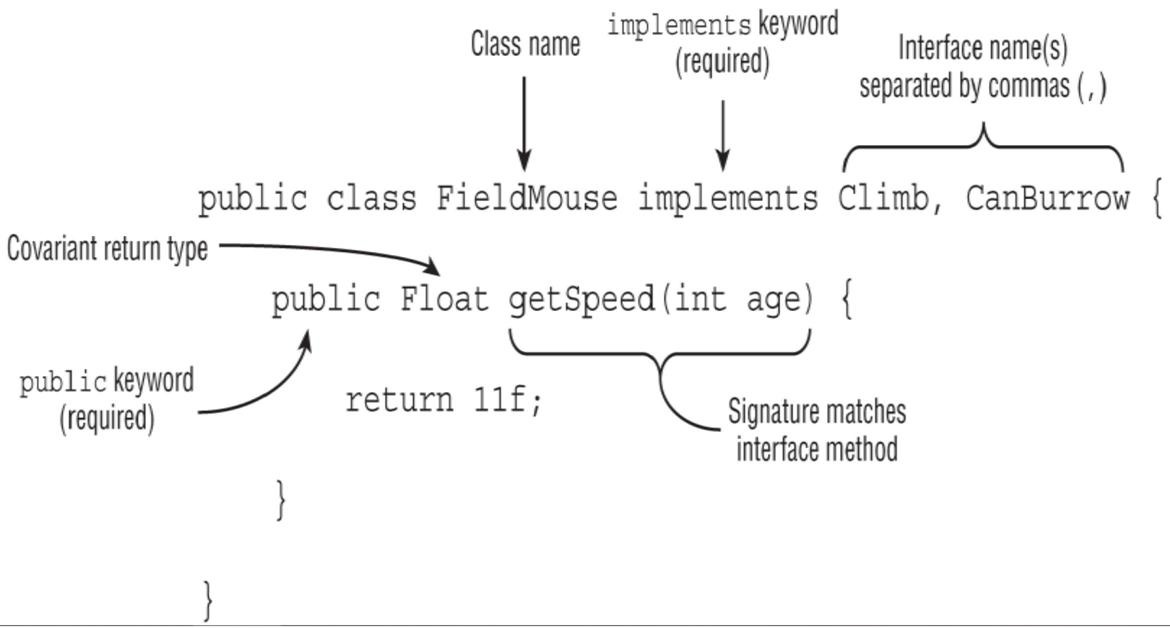
If you come across a question on the exam in which a class or method is marked `abstract`, make sure the class is properly implemented before attempting to solve the problem.

DEFINING AN INTERFACE



```
public abstract interface CanBurrow {

    public abstract Float getSpeed(int age);

    public static final int MINIMUM_DEPTH = 2;

}
```

Implementing an interface


Class name
implements keyword (required)
Interface name(s) separated by commas (,)
public class FieldMouse implements Climb, CanBurrow {
Covariant return type
public Float getSpeed(int age) {
public keyword (required)
return 11f;
Signature matches interface method
}
}

The following list includes the implicit modifiers for interfaces that you need to know for the exam:

1. Interfaces are assumed to be `abstract`.
2. Interface variables are assumed to be `public`, `static`, and `final`.
3. Interface methods without a body are assumed to be `abstract` and `public`.

When working with class members, omitting the access modifier indicates default (package-private) access. When working with interface members, though, the lack of access modifier always indicates `public` access.

Note:

```
interface A {
int m();
}
interface B{
void m();
}

interface C extends A,B{} // DOES NOT COMPILE
abstract class D implements A,B{} // DOES NOT COMPILE
```

here method m() have different return type and compiler cannot decide which to inherit, It is the equivalent of trying to define two methods in the same class with the same signature and different return types.

if return type are covariant (like string and CharSequence) we can override one method m() with return type String, it will override both method m().

valid inheritance:
An interface may extend any number of interfaces and, in doing so, inherits their abstract methods. An interface cannot extend a class, nor can a class extend an interface. A class may implement any number of interfaces.

```
class1 extends class2
interface1 extends interface2, interface3, ...
class1 implements interface2, interface3, ...
```

Interfaces and the *instanceof* Operator (left to add pg 659)

REVIEWING INTERFACE RULES

Interface Definition Rules

1. Interfaces cannot be instantiated.
2. All top-level types, including interfaces, cannot be marked `protected` or `private`.
3. Interfaces are assumed to be `abstract` and cannot be marked `final`.
4. Interfaces may include zero or more abstract methods.
5. An interface can extend any number of interfaces.
6. An interface reference may be cast to any reference that inherits the interface, although this may produce an exception at runtime if the classes aren't related.
7. The compiler will only report an unrelated type error for an `instanceof` operation with an interface on the right side if the reference on the left side is a `final` class that does not inherit the interface.

8. An interface method with a body must be marked `default`, `private`, `static`, or `private static` (covered when studying for the 1Z0-816 exam).

Abstract Interface Method Rules

1. Abstract methods can be defined only in abstract classes or interfaces.
2. Abstract methods cannot be declared `private` or `final` or `static`.
3. Abstract methods must not provide a method body/implementation in the abstract class in which is it declared.
4. Implementing an abstract method in a subclass follows the same rules for overriding a method, including covariant return types, exception declarations, etc.
5. Interface methods without a body are assumed to be `abstract` and `public`.
6. Using a modifier that conflicts with one of these implicit modifiers will result in a compiler error.

The first four rules for abstract methods, whether they be defined in abstract classes or interfaces, are exactly the same! The only new rule you need to learn for interfaces is the last one.
Finally, there are two rules to remember for interface variables.

Interface Variables Rules

1. Interface variables are assumed to be `public`, `static`, and `final`.
2. Because interface variables are marked `final`, they must be initialized with a value when they are declared.

A top-level class or interface is one that is not
defined within another class declaration, while an inner class
or interface is one defined within another class. Inner classes
can be marked `public`, `protected`, package-private, or `private`.

Casting In Interfaces (IMP)

In classes, compiler can detect that the classes are of compatible type or not, but in interfaces compiler allows to cast unrelated Interface reference type to other unrelated interface type variable (since there maybe a subclass that implement that other interface) and give a castclass exception at run time.

But if a class X is marked final and implements A, when we try to assign object of X to other unrelated Interface B, then it will not compile because compiler knows that X is final and there is no subclass of X that implements B.

Therefore, using <u>instanceof</u> operator with two unrelated classes will give compile time error, but with two unrelated interface it will not due to the above-mentioned limitations.

Chapter 10: Exceptions

Methods of available in exception class

- e.getMessage() // get String msg about the exception
- e.printStackTrace()
- e.getSuppressed()
- Throwable.getCause() //used to get the reason that caused the exception, cause means what is passed as argument when exception is thrown.

e.g:

Exception e = new Exception("cause") // e.getCause() -> cause

Exception e = new Exception() // e.getCause() -> null

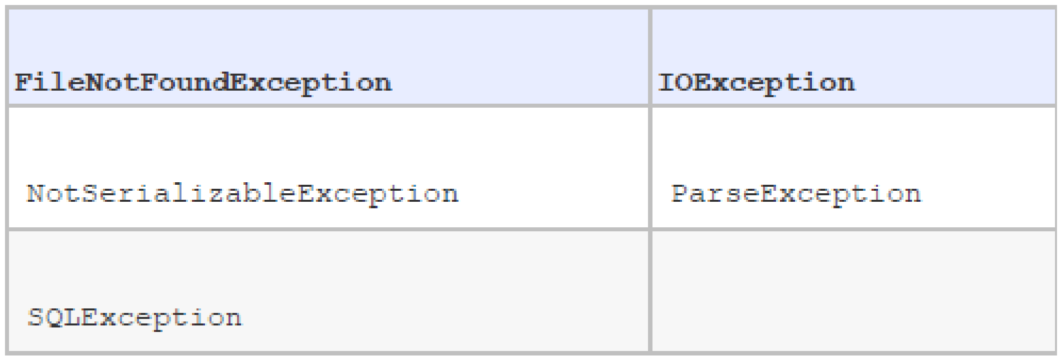
Exception e = new Exception(new IOException()) // e.getCause() -> IOException

Checked Exceptions

A *checked exception* is an exception that must be declared or handled by the application code where it is thrown. In Java, checked exceptions all inherit `Exception` but not `RuntimeException`.

Checked exceptions also include any class that inherits `Throwable`, but not `Error` or `RuntimeException`.

TABLE 16.2 Checked exceptions

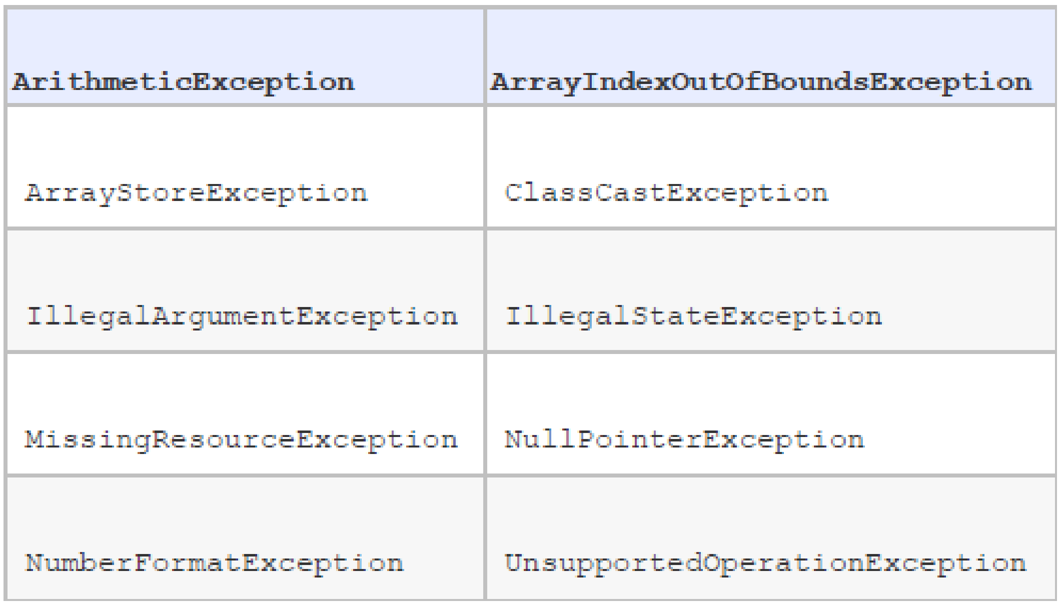


FileNotFoundException	IOException
NotSerializableException	ParseException
SQLException	

Unchecked Exceptions

An *unchecked exception* is any exception that does not need to be declared or handled by the application code where it is thrown. Unchecked exceptions are often referred to as runtime exceptions, although in Java, unchecked exceptions include any class that inherits `RuntimeException` or `Error`.

TABLE 16.1 Unchecked exceptions



ArithmeticException	ArrayIndexOutOfBoundsException
ArrayStoreException	ClassCastException
IllegalArgumentException	IllegalStateException
MissingResourceException	NullPointerException
NumberFormatException	UnsupportedOperationException

NOTE: A runtime (unchecked) exception is a specific type of exception. All exceptions occur at the time that the program is run. (The alternative is compile time, which would be a compiler error.) People don't refer to them as "run time" exceptions because that would be too easy to confuse with runtime! When you see *runtime*, it means unchecked.

THROW VS. *THROWS*

Anytime you see `throw` or `throws` on the exam, make sure the correct one is being used. The `throw` keyword is used as a statement inside a code block to throw a new exception or rethrow an existing exception, while the `throws` keyword is used only at the end of a method declaration to indicate what exceptions it supports.

```
throw RuntimeException(); // DOES NOT COMPILE, new required
```

ALWAYS First check throw and throws are used correctly otherwise compile error

like, return, break, continue, Anything after throw statement is unreachable code and it give compile error.

Chaining multiple catch block

A rule exists for the order of the `catch` blocks. Java looks at them in the order they appear. If it is impossible for one of the `catch` blocks to be executed, a compiler error about unreachable code occurs. For example, this happens when a superclass `catch` block appears before a subclass `catch` block.

remember that at most one `catch` block will run, and it will be the first `catch` block that can handle it. Also, remember that an exception defined by the `catch` statement is only in scope for that `catch` block.

Multi-catch

1. A multi-catch block allows multiple exception types to be caught by the same `catch` block.
2. the syntax of multi-catch. It's like a regular `catch` clause, except two or more exception types are specified separated by a pipe (`|`).
3. Notice how there is only one variable name in the `catch` clause. Java is saying that the ariable named `e` can be of type `Exception1` or `Exception2`.
4. Java intends multi-catch to be used for exceptions that aren't related, and it prevents you from specifying redundant types in a multi-catch.

```
try {
     throw new IOException();
} catch (FileNotFoundException | IOException p) {} //DOES
NOT COMPILE
```

Since `FileNotFoundException` is a subclass of `IOException`, this code will not compile. A multi-catch block follows similar rules as chaining `catch` blocks together that you saw in the previous section. For example, both trigger compiler errors when they encounter unreachable code or duplicate exceptions being caught. The one difference between multi-catch blocks and chaining `catch` blocks is that order does not matter for a multicatch block within a single `catch` expression.

Remember that there can be only one exception variable per `catch` block.
You can't list the same exception type more than once in the same `try` statement, just like with "regular" `catch` blocks.

Finally Block

If an exception is thrown, the `finally` block is run after the `catch` block. If no exception is thrown, the `finally` block is run after the `try` block completes.
If a `try` statement with a `finally` block is entered, then the `finally` block will always be executed, regardless of whether the code completes successfully.

```
12: int goHome() {
13: try {
14:       // Optionally throw an exception here
15:       System.out.print("1");
16:       return -1;
17:  } catch (Exception e) {
18:       System.out.print("2");
19:       return -2;
20:  } finally {
21:       System.out.print("3");
22:       return -3;
23:  }
24:}
```

If an exception is not thrown on line 14, then the line 15 will be executed, printing `1`. Before the method returns, though, the `finally` block is executed, printing `3`. If an exception is thrown, then lines 15–16 will be skipped, and lines 17–19 will be executed, printing `2`, followed by `3` from the `finally` block. While the first value printed may differ, the method always prints `3` last since it's in the `finally` block.

What is the return value of the `goHome()` method? In this case, it's always `-3`. Because the `finally` block is executed shortly before the method completes, it interrupts the `return` statement from inside both the `try` and `catch` blocks.

while a `finally` block will always be executed, it may not finish.
the first line of finally block is guaranteed to execute.
one edge case: When `System.exit()` is called in the `try` or `catch` block, the `finally` block does not run.

try-with-resources

IMPLICIT *FINALLY* BLOCKS

Behind the scenes, the compiler replaces a try-with-resources block with a `try` and `finally` block. We refer to this "hidden" `finally` block as an implicit `finally` block since it is created and used by the compiler automatically. You can still create a programmer-defined `finally` block when using a try-with-resources statement; just be aware that the implicit one will be called first.

In a try-with-resources statement, implicit finally block is run after the try block and before the user added catch and finally. the resource will be closed at the completion of the `try` block, before any declared `catch` or `finally` blocks execute.

one or more resources can be opened in the `try` clause. When there are multiple resources opened, they are closed in the *reverse* order from which they were created. Also, notice that parentheses are used to list those resources, and semicolons are used to separate the declarations.

Remember that only a try-with-resources statement is permitted to omit both the `catch` and `finally` blocks. A traditional `try` statement must have either or both. You can easily distinguish between the two by the presence of parentheses, `()`, after the `try` keyword.
Note:
You can't just put any random class in a try-with-resources statement. Java requires classes used in a trywith- resources implement the `AutoCloseable` interface, which includes a `void close()` method.

Since `Closeable` extends `AutoCloseable`, they are both supported in try - with - resources statements. The only difference between the two is that `Closeable`'s `close()` method declares `IOException`, while `AutoCloseable`'s `close()` method declares `Exception`.

Scope of Try-with-Resources

The resources created in the `try` clause are in scope only within the `try` block. This is another way to remember that the implicit `finally` runs before any `catch`/`finally` blocks that you code yourself. The implicit close has run already, and the resource is no longer available.

Declaring Resources

While try-with-resources does support declaring multiple variables, each variable must be declared in a separate statement. For example, the following do not compile:

```
try (MyFile is = new MyFile (1),os = new MyFile (2)) {
} // DOES NOT COMPILE comma is used and no type is provided

try (MyFile ab = new MyFile (1), MyFile cd = new MyFile (2)) {
} // DOES NOT COMPILE comma is used
```

Each resource must include the data type and be separated by a semicolon (`;`).
You can declare a resource using `var` as the data type in a trywith-resources statement, since resources are local variables.

it is possible to use resources declared prior to the try-with-resources statement, provided they are marked `final` or effectively final.
eg:

```
final var bR = new MyFileReader("4");
MyFileReader mR = new MyFileReader("5");

try (bR; var tvReader = new MyFileReader("6"); mR) {}
```

Lost Exception

only the last exception to be thrown matters others are lost/forgotten exceptions.

```
try {
        throw new RuntimeException();
  } catch (RuntimeException e) {
        throw new RuntimeException();
  } finally {
        throw new Exception();
}
```

SUPPRESSED EXCEPTIONS

it is possible **only in** try-with-resources
when a exception is thrown in try block it is called primary exception, then implicit finally is executed if it also throw one/more exception(s) then these exceptions are called suppressed exceptions and the compiler will only match the primary exception with the following catch blocks.
if primary exception is not matiching any of the catch blocks then it is thrown to the caller, suppressed exceptions are never used to match exceptions in the catch block
e.getSuppressed() is used to get a List of all suppressed exceptions.

Calling Methods That Throw Exceptions

if a method don't actually throw an exception; it just declared that it could. This is enough for the compiler to require the caller to handle or declare the exception.

If a method is called in try block and it declare a runtime exception then the catch block must not contain any checked exceptions since it is not possible for that method to throw any other exception then runtime exception.

If try and finally is there and no catch block, it is considered as not handled and method must declare it.

Always check if it is possible for the called method in try block to throw all type of exceptions in all catch blocks, if not, compile error: unreachable code.

Nested try-catch

in nested try-catch check is the exception variable "e" is not used twice

Creating Custom Exceptions

When creating your own exception, you need to decide whether it should be a checked or unchecked exception. While you can extend any exception class, it is most common to extend `Exception` (for checked) or `RuntimeException` (for unchecked).

class MyException<T> extends Exception{ } // not compile coz generic parameter

interface MyException extends Exception{ } // not a valid exception class.

Adding custom constructor in MyException class

```
public MyException(){

        super();

}

public MyException(Exception e){

        super(e);

}

public MyException(String msg){

        super(msg);

}
```

Exam Essentials

Understand the various types of exceptions. All exceptions are subclasses of `java.lang.Throwable`. Subclasses of `java.lang.Error` should never be caught. Only subclasses of `java.lang.Exception` should be handled in application code.

Differentiate between checked and unchecked exceptions. Unchecked exceptions do not need to be caught or handled and are subclasses of `java.lang.RuntimeException` and `java.lang.Error`. All other subclasses of `java.lang.Exception` are checked exceptions and must be handled or declared.

Understand the flow of a `try` statement. A `try` statement must have a `catch` or a `finally` block. Multiple `catch` blocks can be chained together, provided no superclass exception type appears in an earlier `catch` block than its subclass. A multicatch expression may be used to handle multiple exceptions in the same `catch` block, provided one exception is not a subclass of another. The `finally` block runs last regardless of whether an exception is thrown.

Be able to follow the order of a try-with-resources statement. A try-with-resources statement is a special type of `try` block in which one or more resources are declared and automatically closed in the reverse order of which they are declared. It can be used with or without a `catch` or `finally` block, with the implicit `finally` block always executed first.

Identify whether an exception is thrown by the programmer or the JVM. `IllegalArgumentException` and `NumberFormatException` are commonly thrown by the programmer. Most of the other unchecked exceptions are typically thrown by the JVM or built-in Java libraries.

Write methods that declare exceptions. The `throws` keyword is used in a method declaration to indicate an exception might be thrown. When overriding a method, the method is allowed to throw fewer or narrower checked exceptions than the original version.

Recognize when to use throw versus throws. The `throw` keyword is used when you actually want to throw an exception—for example, `throw new RuntimeException()`. The `throws` keyword is used in a method declaration.

Chapter 11: Modules

check Chapter 17: Modular Applications, all notes there

Chapter 12: Java Fundamentals

Enums

Enums are implicitly static therefore can only be defined in top-level types and static nested classes.

To create an enum, use the `enum` keyword instead of the `class` or `interface` keyword. Then list all of the valid types for that enum.

```
public enum Season {
WINTER, SPRING, SUMMER, FALL // case sensitive
}

Season s = Season.SUMMER;
System.out.println(Season.SUMMER); // SUMMER
System.out.println(s == Season.SUMMER); // true
```

An enum provides many methods to access the values.

```
for(Season season: Season.values()) {
System.out.println(season.name() + " " + season.ordinal());
}

WINTER 0
SPRING 1
SUMMER 2
FALL 3

if ( Season.SUMMER == 2) {} // DOES NOT COMPILE
```

One thing that you can't do is extend an enum.

```
public enum ExtendedSeason extends Season { } // DOES NOT
COMPILE
```

Enums can be used in `switch` statements.
But the case values should be the enum constants nothing else, like in season there is no RAINY, so it cannot be used in any case value
Also not "RAINY", 123, etc.

ADDING CONSTRUCTORS to enum

Enum constructors are implicit private.

```
public enum Season {
  WINTER("Low"), SPRING("Medium"), SUMMER("High"),
  FALL("Medium");
  private final String expectedVisitors;
  private Season(String expectedVisitors) {
       this.expectedVisitors = expectedVisitors;
  }
  public void printExpectedVisitors() {
       System.out.println(expectedVisitors);
  }
}
```

Each enum value can override the default method in enum.

```
public enum Season {
     WINTER {
          public String getHours() { return "10am-3pm"; }
     },
     SUMMER {
          public String getHours() { return "9am-7pm"; }
     },
     SPRING, FALL;
     public String getHours() { return "9am-5pm"; }
}
```

Enums can have abstract method, if a method is abstract all the enum constant must override the abstract method.

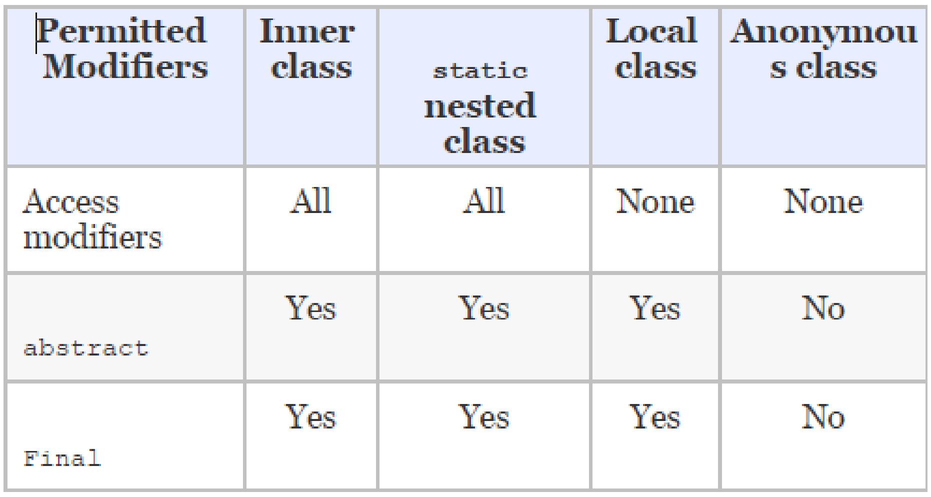
```
public enum Season {
     WINTER {
          public String getHours() { return "10am-3pm"; }
     },
     SPRING {
          public String getHours() { return "9am-5pm"; }
     },

     SUMMER {
          public String getHours() { return "9am-7pm"; }
     },
     FALL {
          public String getHours() { return "9am-5pm"; }
     };
     public abstract String getHours();
}
```

Nested Classes

1. *Inner class*: A non - `static` type defined at the member level of a class
2. *Static nested class:* A `static` type defined at the member level of a class
3. *Local class*: A class defined within a method body
4. *Anonymous class*: A special case of a local class that does not have a name

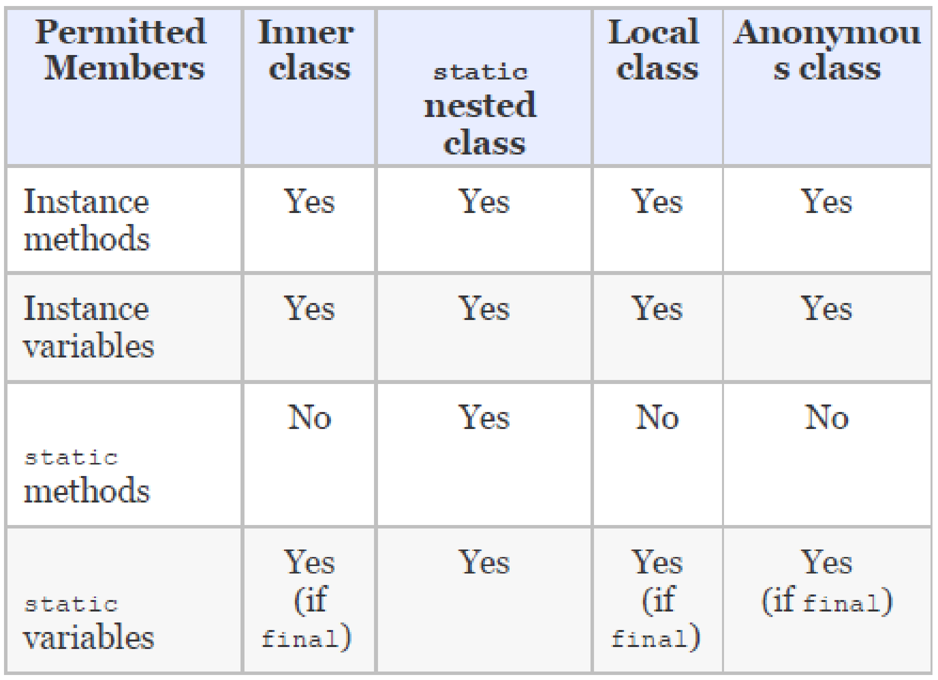
Nested Classes Rules.
Note:
when there is a non static inner class, the object of inner is created as following
new Outter.new Inner()
new Outter().new Inner() // DOES NOT COMPILE

TABLE 12.1 Modifiers in nested classes



Permitted Modifiers	**Inner class**	`static` **nested class**	**Local class**	**Anonymous class**
Access modifiers	All	All	None	None
`abstract`	Yes	Yes	Yes	No
`Final`	Yes	Yes	Yes	No
static	No	Yes	No	No

TABLE 12.2 Members in nested classes



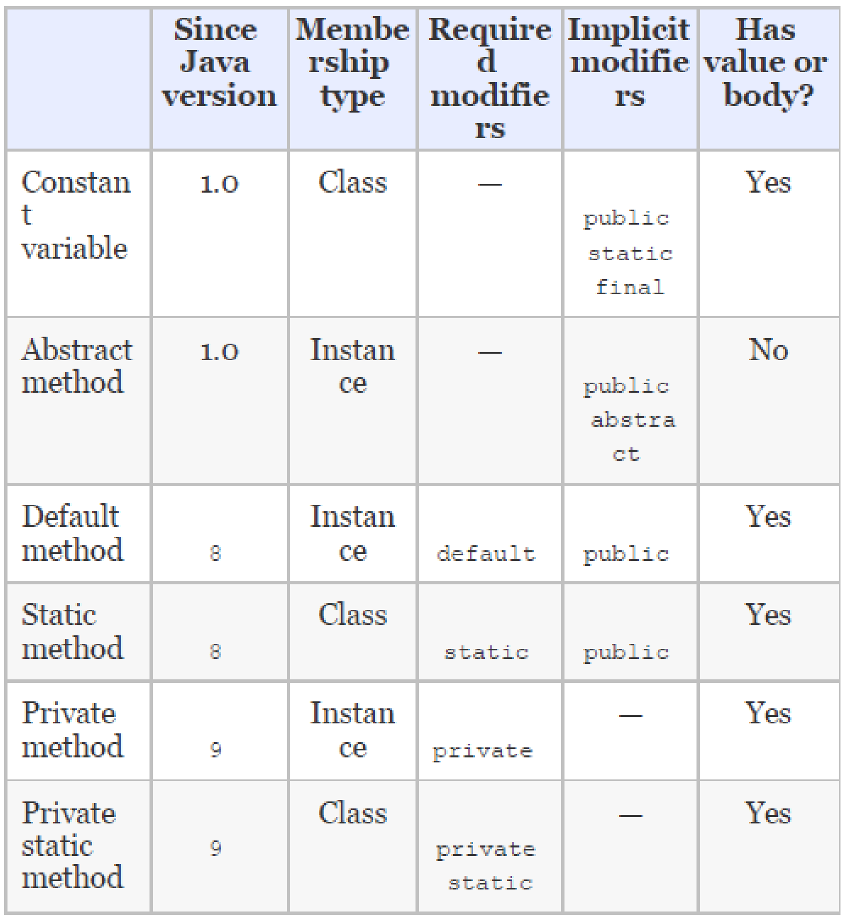
Permitted Members	Inner class	`static` nested class	Local class	Anonymous class
Instance methods	Yes	Yes	Yes	Yes
Instance variables	Yes	Yes	Yes	Yes
`static` methods	No	Yes	No	No
`static` variables	Yes (if `final`)	Yes	Yes (if `final`)	Yes (if `final`)

Nested class access rules

	Inner class	Static nested class	Local class	Anonymous class
Can extend any class or implement any number of interfaces	YES	YES	YES	No—must have exactly one superclass or one interface
Can access instance members of enclosing class without a reference	YES	NO	YES (if declared in an instance method)	YES (if declared in an instance method)
Can access local variables of enclosing method	NA	NA	Yes (if `final` or effectively final)	Yes (if `final` or effectively final)

Interface Members

TABLE 12.4 Interface member types



	Since Java version	Membership type	Required modifiers	Implicit modifiers	Has value or body?
Constant variable	1.0	Class	—	`public` `static` `final`	Yes
Abstract method	1.0	Instance	—	`public` `abstract`	No
Default method	8	Instance	`default`	`public`	Yes
Static method	8	Class	`static`	`public`	Yes
Private method	9	Instance	`private`	—	Yes
Private static method	9	Class	`private` `static`	—	Yes

DEFAULT INTERFACE METHOD DEFINITION RULES

1. A `default` method may be declared only within an interface.
2. A `default` method must be marked with the `default` keyword and include a method body.
3. A `default` method is assumed to be `public`.
4. A `default` method cannot be marked `abstract`, `final`, or `static`.
5. A `default` method may be overridden by a class that implements the interface.
6. If a class inherits two or more `default` methods with the same method signature, then the class must override the method. (if both the default method have same signature but different return type that are not covariant the class/interface implementing/extending such interfaces will not compile).

7. Calling a Hidden *default* Method: InterfaceName.super.default_method_name();

Static Interface Method Definition Rules

1. A `static` method must be marked with the `static` keyword and include a method body.
2. A `static` method without an access modifier is assumed to be `public`.
3. A `static` method cannot be marked `abstract` or `final`.
4. A `static` method is not inherited and cannot be accessed in a class implementing the interface without a reference to the interface name.
5. Calling a Static method in subtype: InterfaceName.static_method_name();

Private Interface Method Definition Rules

1. A `private` interface method must be marked with the `private` modifier and include a method body.
2. A `private` interface method may be called only by `default` and `private` (non - `static`) methods within the interface definition.

Private Static Interface Method Definition Rules

1. A `private static` method must be marked with the `private` and `static` modifiers and include a method body.
2. A `private static` interface method may be called only by other methods within the interface definition.

Interface member access

	Accessible from `default` and `private` methods within the interface definition?	Accessible from `static` methods within the interface definition ?	Accessible From instance methods implementing or extending the interface?	Accessible outside the interface without an instance of interface?
Constant variable	**YES**	**YES**	**YES**	**YES**
Abstract method	**YES**	**NO**	**YES**	**NO**
Default method	**YES**	**NO**	**YES**	**NO**
Private method	**YES**	**NO**	**NO**	**NO**
Static method	**YES**	**YES**	**YES**	**YES**
Private static method	**YES**	**YES**	**NO**	**NO**

Functional interface
Determine whether an interface is a functional interface.
Use the single abstract method (SAM) rule to determine whether an interface is a functional interface. Other interface method types (`default`, `private`, `static`, and `private static`) do not count toward the single abstract method count, nor do any `public` methods with signatures found in `Object`.

```
@FunctionalInterface
//It is Functional interface
public interface Sprint {
     public void sprint(int speed);
}
```

```
//It is Functional interface
public interface Dash extends Sprint {}
```

//It is not a Functional interface(total 2 abstract method)
```
public interface Skip extends Sprint {
     void skip();
}
```

//It is not a Functional interface(no abstract method)
```
public interface Sleep {
     private void snore() {}
     default int getZzz() { return 1; }
}
```

//It is Functional interface
```
public interface Climb {
     void reach();
     default void fall() {}
     static int getBackUp() { return 100; }
     private static boolean checkHeight() { return true; }
}
```

Methods in Object class

- `String toString()`
- `boolean equals(Object)`
- `int hashCode()`

Note:

Since Java assumes all classes extend from `Object`, you also cannot declare an interface method that is incompatible with `Object`. For example, declaring an abstract method `int toString()` in an interface would not compile since `Object`'s version of the method returns a `String`.

If a functional interface includes an abstract method with the same signature as a `public` method found in `Object`, then those methods do not count toward the single abstract method test.
eg:
//It is not a Functional interface. Since `toString()` is a `public` method implemented in `Object`, it does not count toward the single abstract method test.
```
public interface Soar {
     abstract String toString();
}
```

//It is Functional interface

```
public interface Dive {
     String toString();
     public boolean equals(Object o);
     public abstract int hashCode();
     public void dive();
}
```

//It is not a Functional interface (it has 2 abstract method, rest and equals)

```
public interface Hibernate {
     String toString();
     public boolean equals(Hibernate o);
     public abstract int hashCode();
     public void rest();
}
```

TABLE 15.1 Common functional interfaces

Functional interface	Return type	Method name	# of parameters
`Supplier<T>`	`T`	`get()`	0
`Consumer<T>`	`void`	`accept(T)`	1 `(T)`
`BiConsumer<T, U>`	`void`	`accept(T,U)`	2 `(T, U)`
`Predicate<T>`	`boolean`	`test(T)`	1 `(T)`
`BiPredicate<T, U>`	`boolean`	`test(T,U)`	2 `(T, U)`
`Function<T, R>`	`R`	`apply(T)`	1 `(T)`
`BiFunction<T, U, R>`	`R`	`apply(T,U)`	2 `(T, U)`
`UnaryOperator<T>`	`T`	`apply(T)`	1 `(T)`
`BinaryOperator<T>`	`T`	`apply(T,T)`	2 `(T, T)`

Lambda Expression

Rules of defining a Lambda Expression from kamal sir notes **Book 2**

Functional interfaces are available in **java.util.function**

1. **java.util.function.Function**

   ```
   public interface Function<T,R>{
           public <R> apply(T t);
   }
   eg:
   Function<Integer,Double> function = a -> a/2
   function.apply(12); // 6.0
   ```

2. **java.util.function.Predicate**

   ```
   public interface Predicate<T>{
           Boolean test(T t);
   }
   eg:
   Predicate<Object> predicate = a -> a != null;
   predicate.test(“hello”); //false
   predicate.test(null); //true
   ```

3. **java.util.function.Consumer**

   ```
   public interface Consumer<T>{
           void accept(T t);
           // generally used to print something
   }
   eg:
   ```

   ```
   Consumer<String> consumer = x -> System.out.println(x);
   consumer.accept("Hello World"); // Hello World
   ```

4. **java.util.function.Supplier**

   ```
   public interface Supplier<T>{
           public <T> get();
           //generally used to create a object and return
   }
   eg:
   Supplier<Integer> number = () -> 42;
   System.out.println(number.get()); // 42
   ```

5. **java.util.function.Comparator**

   ```
   public interface Comparator<T>{
           public int compare(T t1, T t2);
   }
   eg:
   Comparator<Integer,Integer> comparator = (a,b) -> a-b;
   System.out.println(comparator.compare(100,50)); // 50
   ```

```
(a, b) -> { int a = 0; return 5;} // DOES NOT COMPILE
```

a is already defined in the method scope as a method parameter hence cannot be declared again.

Restrictions on Using var in the Parameter List

While you can use `var` inside a lambda parameter list, there is a rule you need to be aware of. If `var` is used for one of the types in the parameter list, then it must be used for all parameters in the list.

TABLE 6.4 Rules for accessing a variable from a lambda body inside a method

Variable type	Rule
Instance variable	Allowed
Static variable	Allowed
Local variable	Allowed if effectively final
Method parameter	Allowed if effectively final
Lambda parameter	Allowed

effectively final, This means that the value of a variable doesn't change after it is set, regardless of whether it is explicitly marked as `final`.

Determine whether a variable can be used in a lambda body. Local variables and method parameters must be effectively final to be referenced. This means the code must compile if you were to add the `final` keyword to these variables. Instance and class variables are always allowed.

Chapter 13: Annotations

IMP OBSERVATIONS

Elements of annotations can be:

- primitive
- String
- an Enum
- another Annotation
- Class
- an Array of any of the above

NOTE:

1. an element declared in an annotation must use parentheses after its name.
2. no multi-dimensional array (like String[][])
3. NO wrapper classes
4. constants are declared using “=” sign, and cannot be passed as a parameter to the annotation because they are constant.
5. if a final variable is declared but not initialised, It give compile error, it must be initialised in the same line.
6. annotations cannot declare Constructor or any method.
7. method can be annotated with only the annotations that do not take arguments. // not sure.
8. By default, annotations are not present at runtime
9. A default annotation element value must be a non-null constant expression. eg: String name() default null; // does not compile.

TABLE 13.5 Understanding common annotations

Annotation	Marker annotation	Type of `value()`	Optional members
`@Override`	Yes	—	—
`@FunctionalInterface`	Yes	—	—
`@Deprecated`	No	—	`String since()` `boolean forRemoval()`
`@SuppressWarnings`	No	`String[]`	—
`@SafeVarargs`	Yes	—	—

TABLE 13.6 Applying common annotations

Annotation	Applies to	Compiler error when
`@Override`	Methods	Method signature does not match the signature of an inherited method
`@FunctionalInterface`	Interfaces	Interface does not contain a single abstract method
`@Deprecated`	Most Java declarations	—
`@SuppressWarnings`	Most Java declarations	—
`@SafeVarargs`	Methods, constructors	Method or constructor does not contain a varargs parameter or is applied to a method not marked `private`, `static`, or `final`

Exam Essentials

Be able to declare annotations with required elements, optional elements, and variables. An annotation is declared with the `@interface` type. It may include elements and `public static final` constant variables. If it does not include any elements, then it is a marker annotation. Optional elements are specified with a `default` keyword and value, while required elements are those specified without one.

Be able to identify where annotations can be applied. An annotation is applied using the at (`@`) symbol, followed by the annotation name. Annotations must include a value for each required element and can be applied to types, methods, constructors, and variables. They can also be used in cast operations, lambda expressions, or inside type declarations.

Understand how to apply an annotation without an element name. If an annotation contains an element named `value()` and does not contain any other elements that are required, then it can be used without the *`elementName`*. For it to be used properly, no other values may be passed.

Understand how to apply an annotation with a singleelement array. If one of the annotation elements is a primitive array and the array is passed a single value, then the annotation value may be written without the array braces (`{}`).

Apply built - in annotations to other annotations. Java includes a number of annotations that apply to annotation declarations. The `@Target` annotation allows you to specify where an annotation can and cannot be used. The `@Retention` annotation allows you to specify at what level the annotation metadata is kept or discarded. `@Documented` is a marker annotation that allows you to specify whether annotation information is included in the generated documentation. `@Inherited` is another marker annotation that determines whether annotations are inherited from super types. The `@Repeatable` annotation allows you to list an annotation more than once on a single declaration. It requires a second containing type annotation to be declared.

Apply common annotations to various Java types. Java includes many built - in annotations that apply to classes, methods, variables, and expressions. The `@Override` annotation is used to indicate that a method is overriding an inherited method. The `@FunctionalInterface` annotation confirms that an interface contains exactly one abstract method. Marking a type `@Deprecated` means that the compiler will generate a depreciation warning when it is referenced. Adding `@SuppressWarnings` with a set of values to a declaration causes the compiler to ignore the set of specified warnings. Adding `@SafeVarargs` on a constructor or `private`, `static`, or `final` method instructs other developers that no unsafe operations will be performed on its varargs parameter. While all of these annotations are optional, they are quite useful and improve the quality of code when used.

Chapter 14: Generics and Collections

READ MODULE NO. 23 – 27 FOR COLLECTIONS

IMP OBSERVATIONS

int[] arr = new int[1] {5}; //not allowed, when providing elements size is automatically infered

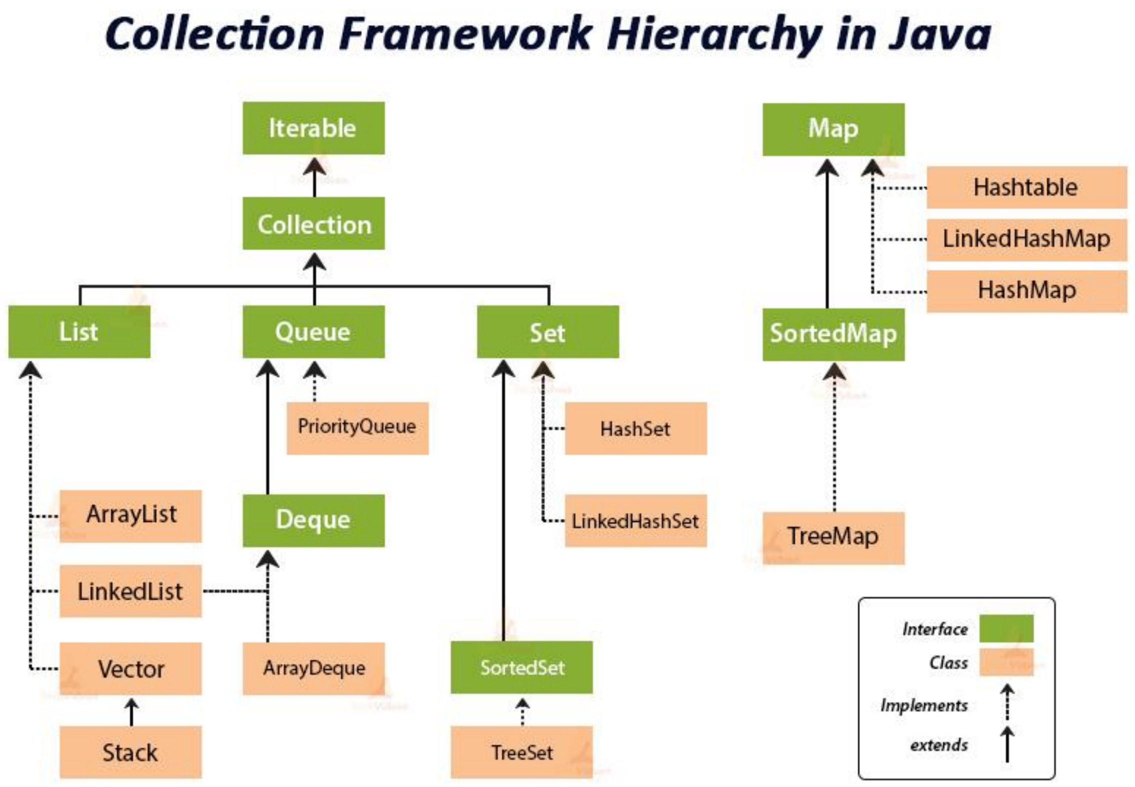
var[] arr = new int[] {1,2} // not allowed
var arr = new int[] {1,2} // allowed

List<String> strings = new ArrayList<?>(); // not compile, wildcard not allowed on right side

set.iterator().next() will give first element

in DeQueue/LinkedList - push() head pe append krta hai and offer()/add() back se daalta hai

Generics



Summary

A method reference is a compact syntax for writing lambdas that refer to methods. There are four types: `static` methods, instance methods on a particular object, instance methods on a parameter, and constructor references.

The diamond operator (<>) is used to tell Java that the generic type matches the declaration without specifying it again. The diamond operator can be used for local variables or instance variables as well as one - line declarations.

The `Arrays` and `Collections` classes have methods for `sort()` and `binarySearch()`. Both take an optional `Comparator` parameter. It is necessary to use the same sort order for both sorting and searching, so the result is not undefined.

Generics are type parameters for code. To create a class with a generic parameter, add `<T>` after the class name. You can use any name you want for the type parameter. Single uppercase letters are common choices.

Generics allow you to specify wildcards.
`<?>` is an unbounded wildcard that means any type.
`<? extends Object>` is an upper bound that means any type that is `Object` or extends it. `<? extends MyInterface>` means any type that implements `MyInterface`.
`<? super Number>` is a lower bound that means any type that is `Number` or a superclass.
A compiler error results from code that attempts to add an item in a list with an unbounded or upper - bounded wildcard.

Exam Essentials

Translate method references to the "long form" lambda. Be able to convert method references into regular lambda expressions and vice versa. For example, `System.out::print` and `x -> System.out.print(x)` are equivalent. Remember that the order of method parameters is inferred for both based on usage.

Work with convenience methods. The Collections Framework contains many methods such as `contains()`, `forEach()`, and `removeIf()` that you need to know for the exam. There are too many to list in this paragraph for review, so please do review the tables in this chapter.

Differentiate between Comparable and Comparator. Classes that implement `Comparable` are said to have a natural ordering and implement the `compareTo()` method. A class is allowed to have only one natural ordering. A `Comparator` takes two objects in the `compare()` method. Different `Comparator`s can have different sort orders. A `Comparator` is often implemented using a lambda such as `(a, b) -> a.num – b.num`.

Write code using the diamond operator. The diamond operator (`<>`) is used to write more concise code. The type of the generic parameter is inferred from the surrounding code. For example, in `List<String> c = new ArrayList<>()`, the type of the diamond operator is inferred to be `String`.

Chapter 15: Functional Programming

Study MODULE NO. 27 from kamal sir notes.

IMP OBSERVATIONS

In stream questions check if source is there or not.
i.e stream is created properly or not.

normal for loop cannot be used on streams.

Summary

An `Optional<T>` can be empty or store a value. You can check whether it contains a value with `isPresent()` and `get()` the value inside. You can return a different value with `orElse(T t)` or throw an exception with `orElseThrow()`. There are even three methods that take functional interfaces as parameters: `ifPresent(Consumer c)`, `orElseGet(Supplier s)`, and `orElseThrow(Supplier s)`. There are three optional types for primitives: `OptionalDouble`, `OptionalInt`, and `OptionalLong`. These have the methods `getAsDouble()`, `getAsInt()`, and `getAsLong()`, respectively.

A stream pipeline has three parts. The source is required, and it creates the data in the stream. There can be zero or more intermediate operations, which aren't executed until the terminal operation runs. The first stream class we covered was `Stream<T>`, which takes a generic argument `T`. The `Stream<T>` class includes many useful intermediate operations including `filter()`, `map()`, `flatMap()`, and `sorted()`. Examples of terminal operations include `allMatch()`, `count()`, and `forEach()`.

Besides the `Stream<T>` class, there are three primitive streams: `DoubleStream`, `IntStream`, and `LongStream`. In addition to the usual `Stream<T>` methods, `IntStream` and `LongStream` have `range()` and `rangeClosed()`. The call `range(1, 10)` on `IntStream` and `LongStream` creates a stream of the primitives from 1 to 9.

By contrast, `rangeClosed(1, 10)` creates a stream of the primitives from 1 to 10. The primitive streams have math operations including `average()`, `max()`, and `sum()`. They also have `summaryStatistics()` to get many statistics in one call. There are also functional interfaces specific to streams. Except for `BooleanSupplier`, they are all for `double`, `int`, and `long` primitives as well.

You can use a `Collector` to transform a stream into a traditional collection. You can even group fields to create a complex map in one line. Partitioning works the same way as grouping, except that the keys are always `true` and `false`. A partitioned map always has two keys even if the value is empty for the key.
You should review the tables in the chapter. While there's a lot of tables, many share common patterns, making it easier to remember them. You absolutely must memorize Table 15.1. You should memorize Table 15.8 and Table 15.9 but be able to spot incompatibilities, such as type differences, if you can't memorize these two. Finally, remember that streams are lazily evaluated. They take lambdas or method references as parameters, which execute later when the method is run.

Exam Essentials

Identify the correct functional interface given the number of parameters, return type, and method name —and vice versa. The most common functional interfaces are `Supplier`, `Consumer`, `Function`, and `Predicate`. There are also binary versions and primitive versions of many of these methods.
Write code that uses *Optional*. Creating an `Optional` uses `Optional.empty()` or `Optional.of()`. Retrieval frequently uses `isPresent()` and `get()`. Alternatively, there are the functional `ifPresent()` and `orElseGet()` methods.

Recognize which operations cause a stream pipeline to execute. -> Intermediate operations do not run until the terminal operation is encountered. If no terminal operation is

in the pipeline, a `Stream` is returned but not executed. Examples of terminal operations include `collect()`, `forEach()`, `min()`, and `reduce()`.
eg:

```
var cats = new ArrayList<String>();
cats.add("Annie");
cats.add("Ripley");
var stream = cats.stream();
cats.add("KC");
System.out.println(stream.count()); // 3
```

Determine which terminal operations are reductions. Reductions use all elements of the stream in determining the result. The reductions that you need to know are `collect()`, `count()`, `max()`, `min()`, and `reduce()`. A mutable reduction collects into the same object as it goes. The `collect()` method is a mutable reduction.

Write code for common intermediate operations. The `filter()` method returns a `Stream<T>` filtering on a `Predicate<T>`. The `map()` method returns a `Stream` transforming each element of type `T` to another type `R` through a `Function <T,R>`. The `flatMap()` method flattens nested streams into a single level and removes empty streams.

Compare primitive streams to *Stream<T>*. Primitive streams are useful for performing common operations on numeric types including statistics like `average()`, `sum()`, etc. There are three primitive stream classes: `DoubleStream`, `IntStream`, and `LongStream`. There are also three primitive `Optional` classes: `OptionalDouble`, `OptionalInt`, and `OptionalLong`. Aside from `BooleanSupplier`, they all involve the `double`, `int`, or `long` primitives.

Convert primitive stream types to other primitive stream types. Normally when mapping, you just call the *map()* method. When changing the class used for the stream, a different method is needed. To convert to `Stream`, you use `mapToObj()`. To convert to `DoubleStream`, you use `mapToDouble()`.

To convert to `IntStream`, you use `mapToInt()`. To convert to `LongStream`, you use `mapToLong()`.

Use *peek()* to inspect the stream. The `peek()` method is an intermediate operation often used for debugging purposes. It executes a lambda or method reference on the input and passes that same input through the pipeline to the next operator. It is useful for printing out what passes through a certain point in a stream.

Search a stream. The `findFirst()` and `findAny()` methods return a single element from a stream in an `Optional`. The `anyMatch()`, `allMatch()`, and `noneMatch()` methods return a `boolean`. Be careful, because these three can hang if called on an infinite stream with some data. All of these methods are terminal operations.

Sort a stream. The `sorted()` method is an intermediate operation that sorts a stream. There are two versions: the signature with zero parameters that sorts using the natural sort order, and the signature with one parameter that sorts using that `Comparator` as the sort order.

Compare *groupingBy()* and *partitioningBy()*. The `groupingBy()` method is a terminal operation that creates a `Map`. The keys and return types are determined by the parameters you pass. The values in the `Map` are a `Collection` for all the entries that map to that key. The `partitioningBy()` method also returns a `Map`. This time, the keys are `true` and `false`. The values are again a `Collection` of matches. If there are no matches for that `boolean`, the `Collection` is empty.

Chapter 16: Localization // kamal

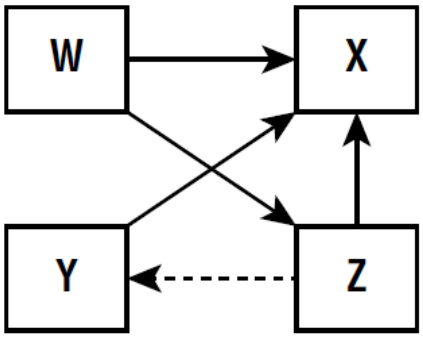
Study MODULE 36 and 37 from kamal sir notes

Chapter 17: Modular Applications

IMP OBSERVATION

Module path contains – Named (module-info.java) and Automatic module (Automatic-module-name in manifest.mf file)
Class path contains – Unnamed modules (even if they have modular jar they are considered unnamed)

- All parts of a modules service must point to the service provider interface
- nothing has a direct dependency on the service provider
- The consumer depends on the service provider interface and service locator, but not the service provider.
- The service locator references the service provider interface directly and the service provider indirectly.
- Eg:



W – Consumer

X – Service locator

Y – Service provider

Z – Service provider Interface

TABLE 11.5 Comparing command-line operations

Description	Syntax

Compile nonmodular code	**`javac -cp`** *`classpath`* `-d` *`directory classesToCompile`*
	
	`javac --class-path` *`classpath`* `-d` *`directory classesToCompile`*
	
	`javac -classpath` *`classpath`* `-d` *`directory classesToCompile`*

Run nonmodular code	**`java -cp`** *`classpath package.className`*
	
	`java -classpath` *`classpath package.className`*
	
	`java --class-path` *`classpath package.className`*

Compile a module	**`javac -p`** *`moduleFolderName`* `-d` *`directory classesToCompileIncludingModuleInfo`*
	
	`javac --module-path` *`moduleFolderName`* `-d directory` *`classesToCompileIncludingModuleInfo`*

Run a module	**`java -p`** *`moduleFolderName`* **`-m`** *`moduleName/package.className`*
	
	`java --module-path` *`moduleFolderName`* **`--module`** *`moduleName/package.className`*

Describe a module	**`java -p`** *`moduleFolderName`* **`-d`** *`moduleName`*
	
	`java --module-path` *`moduleFolderName`* **`--describe-module`** *`moduleName`*

TABLE 11.6 Options you need to know for the exam: `javac`

Option	Description

`-cp <classpath>`
 	
`-classpath <classpath>`
 	
`--class-path <classpath>`
	Location of JARs in a nonmodular program

`-d <dir>`	Directory to place generated class files

`-p <path>`
 	
`--module-path <path>`
	Location of JARs in a modular program

TABLE 11.7 Options you need to know for the exam: `java`

Option	Description

`-p <path>`
 	
`--module-path <path>`
	Location of JARs in a modular program

`-m <name>`
 	
`--module <name>`
	Module name to run

`-d`
 	
`--describe-module`
	Describes the details of a module

`--list-modules`	Lists observable modules without running a program
`--show-module-resolution`	Shows modules when running program

TABLE 11.8 Options you need to know for the exam: `jar`

Option	Description

`-c`
 	
`--create`
	Create a new JAR file

`-v`
 	
`--verbose`
	Prints details when working with JAR files

`-f`
 	
`--file`
	JAR filename

`-C`	Directory containing files to be used to create the JAR

`-d`
 	
`--describe-module`
	Describes the details of a module

TABLE 11.9 Options you need to know for the exam: `jdeps`

Option	Description
`--module-path <path>`	Location of JARs in a modular program

`-s`
 	
`-summary`
	Summarizes output

Summary

The Java Platform Module System organizes code at a higher level than packages. Each module contains one or more packages and a `module-info` file. Advantages of the JPMS include better access control, clearer dependency management, custom runtime images, improved performance, and unique package enforcement.

The process of compiling and running modules uses the `--module-path`, also known as `-p`. Running a module uses the `--module` option, also known as `-m`. The class to run is specified in the format `moduleName/className`.

The `module-info` file supports a number of keywords. The `exports` keyword specifies that a package should be accessible outside the module. It can optionally restrict that export to a specific package. The `requires` keyword is used when a module depends on code in another module. Additionally, `requires transitive` can be used when all modules that require one module should always require another. The `provides` and `uses` keywords are used when sharing and consuming an API. Finally, the `opens` keyword is used for allowing access via reflection.

Both the `java` and `jar` commands can be used to describe the contents of a module. The `java` command can additionally list available modules and show module resolution. The `jdeps` command prints information about packages used in addition to module-level information. Finally, the `jmod` command is used when dealing with files that don't meet the requirements for a JAR.

Exam Essentials

Identify benefits of the Java Platform Module System. Be able to identify benefits of the JPMS from a list such as access control, dependency management, custom runtime images, performance, and unique package enforcement. Also be able to differentiate benefits of the JPMS from benefits of Java as a whole. For example, garbage collection is not a benefit of the JPMS.

Use command-line syntax with modules. Use the command-line options for `javac`, `java`, and `jar`. In particular, understand the module (`-m`) and module path (`-p`) options.

Create basic `module-info` files. Place the `module-info.java` file in the root directory of the module. Know how to code using `exports` to expose a package and how to export to a specific module. Also, know how to code using `requires` and `requires transitive` to declare a dependency on a package or to share that dependency with any modules using the current module.

Identify advanced `module-info` keywords. The `provides` keyword is used when exposing an API. The `uses` keyword is for consuming an API. The `opens` keyword is for allowing the use of reflection.

Display information about modules. The `java` command can describe a module, list available modules, or show the module resolution. The `jar` command can describe a module similar to how the `java` command does. The `jdeps` command prints details about a module and packages. The `jmod` command provides various modes for working with JMOD files rather than JAR files.

TABLE 17.8 Reviewing services

Artifact	Part of the service	Directives required in `module-info.java`
Service provider interface	Yes	`exports`
Service provider	No	`requires` `provides`
Service locator	Yes	`exports` `requires` `uses`
Consumer	No	`requires`

Summary

There are three types of modules. Named modules contain a `module-info.java` file and are on the module path. They can read only from the module path. Automatic modules are also on the module path but have not yet been modularized. They might have an automatic module name set in the manifest. Unnamed modules are on the classpath.

The `java.base` module is most common and is automatically supplied to all modules as a dependency. You do have to be familiar with the full list of modules provided in the JDK. The `jdeps` command provides a list of dependencies that a JAR needs. It can do so on a summary level or detailed level. Additionally, it can specify information about JDK internal modules and suggest replacements.

The two most common migration strategies are top - down and bottom - up migration. Top - down migration starts migrating the module with the most dependencies and places all other modules on the module path. Bottom - up migration starts migrating a module with no dependencies and moves one module to the module path at a time. Both of these strategies require ensuring you do not have any cyclic dependencies since the Java Platform Module System will not allow cyclic dependencies to compile.

Explain top‐down and bottom‐up migration. A topdown migration places all JARs on the module path, making them automatic modules while migrating from top to bottom. A bottom‐up migration leaves all JARs on the classpath, making them unnamed modules while migrating from bottom to top.

A service consists of the service provider interface and service locator. The service provider interface is the API for the service. One or more modules contain the service provider. These modules contain the implementing classes of the service provider interface. The service locator calls `ServiceLoader` to dynamically get any service providers. It can return the results so you can loop through them or get a stream. Finally, the consumer calls the service provider interface.

Code directives for use with services. A service provider implementation must have the `provides` directive to specify what service provider interface it supplies and what class it implements it `with`. The module containing the service locator must have the `uses` directive to specify which service provider implementation it will be looking up.

Exam Essentials

Identify the three types of modules. Named modules are JARs that have been modularized. Unnamed modules have not been modularized. Automatic modules are in between. They are on the module path but do not have a `moduleinfo.java` file.

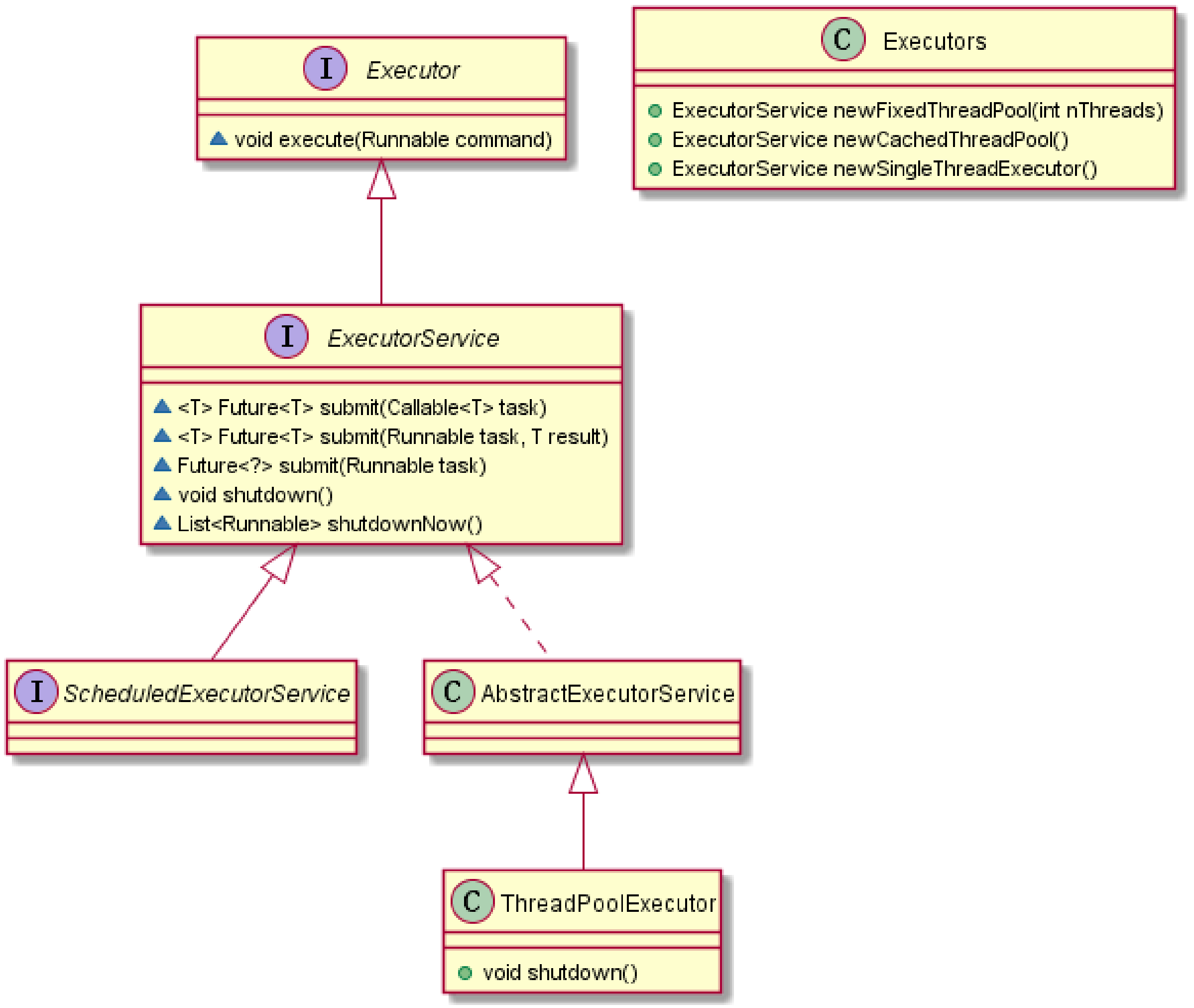
List built‐in JDK modules. The `java.base` module is available to all modules. There are about 20 other modules provided by the JDK that begin with `java.*` and about 30 that begin with `jdk.*`.

Use *jdeps* to list required packages and internal packages. The `-s` flag gives a summary by only including module names. The `--jdk-internals` (`-jdkinternals`) flag provides additional information about unsupported APIs and suggests replacements.

Differentiate the four main parts of a service. A service provider interface declares the interface that a service must implement. The service locator looks up the service, and a consumer calls the service. Finally, a service provider implements the service.

Chapter 18: Concurrency

Study MODULE NO. 38, 39 and 40 from Kamal sir notes



I Executor
▲ void execute(Runnable command)
C Executors
● ExecutorService newFixedThreadPool(int nThreads)
● ExecutorService newCachedThreadPool()
● ExecutorService newSingleThreadExecutor()
I ExecutorService
▲ <T> Future<T> submit(Callable<T> task)
▲ <T> Future<T> submit(Runnable task, T result)
▲ Future<?> submit(Runnable task)
▲ void shutdown()
▲ List<Runnable> shutdownNow()
I ScheduledExecutorService
C AbstractExecutorService
C ThreadPoolExecutor
● void shutdown()

IMP OBSERVATION

- Future<>.get() throws a checked exception.
- Future<?> f1 = service5.submit(() -> System.out.println(“”)); in this case the method does not return anything so the f1 will point to “null” and f1.get() will return null.
- if two methods are synchronous but one is static and other is instance method they are synchronous on different monitor object.

Exam Essentials

Create concurrent tasks with a thread executor service using *Runnable* and *Callable*. An `ExecutorService` creates and manages a single thread or a pool of threads. Instances of `Runnable` and `Callable` can both be submitted to a thread executor and will be completed using the available threads in the service. `Callable` differs from `Runnable` in that `Callable` returns a generic data type and can throw a checked exception. A `ScheduledExecutorService` can be used to schedule tasks at a fixed rate or a fixed interval between executions.

Be able to apply the atomic classes. An atomic operation is one that occurs without interference by another thread. The Concurrency API includes a set of atomic classes that are similar to the primitive classes, except that they ensure that operations on them are performed atomically.

Be able to write thread‐safe code. Thread‐safety is about protecting shared data from concurrent access. A monitor can be used to ensure that only one thread processes a particular section of code at a time. In Java, monitors can be implemented with a `synchronized` block or method or using an instance of `Lock`. `ReentrantLock` has a number of advantages over using a `synchronized` block including the ability to check whether a lock is available without blocking on it, as well as supporting fair acquisition of locks. To achieve synchronization, two threads must synchronize on the same shared object.

Manage a process with a *CyclicBarrier*. The `CyclicBarrier` class can be used to force a set of threads to wait until they are at a certain stage of execution before continuing.

Be able to use the concurrent collection classes. The Concurrency API includes numerous collection classes that include built‐in support for multithreaded processing, such as `ConcurrentHashMap`. It also includes a class `CopyOnWriteArrayList` that creates a copy of its underlying list structure every time it is modified and is useful in highly concurrent environments.

Identify potential threading problems. Deadlock, starvation, and livelock are three threading problems that can occur and result in threads never completing their task. Deadlock occurs when two or more threads are blocked forever. Starvation occurs when a single thread is perpetually denied access to a shared resource. Livelock is a form of starvation where two or more threads are active but conceptually blocked forever. Finally, race conditions occur when two threads execute at the same time, resulting in an unexpected outcome.

Understand the impact of using parallel streams. The Stream API allows for easy creation of parallel streams. Using a parallel stream can cause unexpected results, since the order of operations may no longer be predictable. Some operations, such as `reduce()` and `collect()`, require special consideration to achieve optimal performance when applied to a parallel stream.

Chapter 19 & 20: I/O & NIO.2

Study MODULE NO. 28 and 29 from kamal sir notes

Exam Essentials

Understand files, directories, and streams. Files are records that store data within a persistent storage device, such as a hard disk drive, that is available after the application has finished executing. Files are organized within a file system in directories, which in turn may contain other directories. The root directory is the topmost directory in a file system.

Be able to use the *java.io.File* class. A `java.io.File` instance can be created by passing a path `String` to the `File` constructor. The `File` class includes a number of instance methods for retrieving information about both files and directories. It also includes methods to create/delete files and directories, as well as retrieve a list of files within the directory.

Distinguish between byte and character streams. Streams are either byte streams or character streams. Byte streams operate on binary data and have names that end with `Stream`, while character streams operate on text data and have names that end in `Reader` or `Writer`.

Distinguish between input and output streams. Operating on a stream involves either receiving or sending data. The `InputStream` and `Reader` classes are the topmost abstract classes that receive data, while the `OutputStream` and `Writer` classes are the topmost abstract classes that send data. All I/O output streams covered in this chapter have corresponding input streams, with the exception of `PrintStream` and `PrintWriter`. `PrintStream` is also unique in that it is the only `OutputStream` without the word `Output` in its name.

Distinguish between low - level and high - level streams.

A low - level stream is one that operates directly on the underlying resource, such as a file or network connection. A high - level stream is one that operates on a low - level or other high - level stream to filter data, convert data, or improve performance.

Be able to perform common stream operations. All streams include a `close()` method, which can be invoked automatically with a try - with - resources statement. Input streams include methods to manipulate the stream including `mark()`, `reset()`, and `skip()`. Remember to call `markSupported()` before using `mark()` and `reset()`, as some streams do not support this operation. Output streams include a `flush()` method to force any buffered data to the underlying resource.

Be able to recognize and know how to use various stream classes. Besides the four top - level abstract classes, you should be familiar with the file, buffered, print, and object stream classes. You should also know how to wrap a stream with another stream appropriately.

Understand how to use Java serialization. A class is considered serializable if it implements the `java.io.Serializable` interface and contains instance members that are either serializable or marked `transient`. All Java primitives and the `String` class are serializable. The `ObjectInputStream` and `ObjectOutputStream` classes can be used to read and write a `Serializable` object from and to a stream, respectively.

Be able to interact with the user. Be able to interact with the user using the system streams (`System.out`, `System.err`, and `System.in`) as well as the `Console` class. The `Console` class includes special methods for formatting data and retrieving complex input such as passwords.

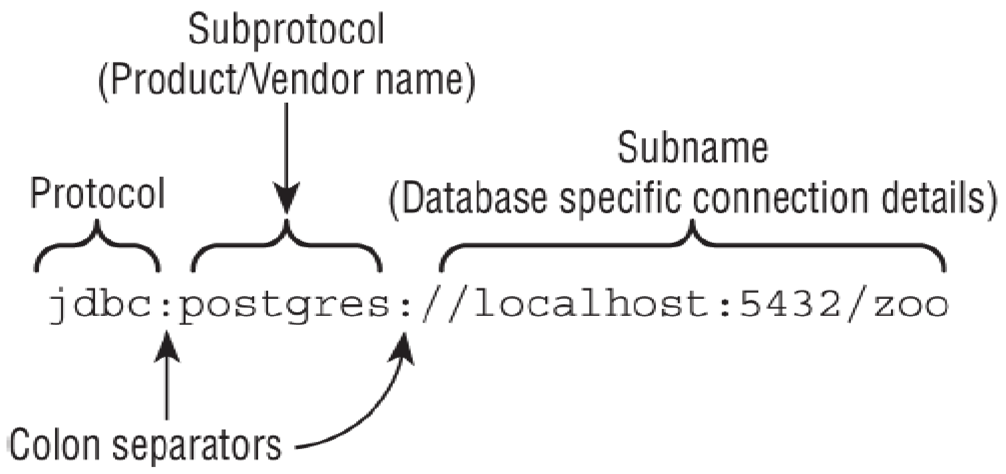
Chapter 21: JDBC

STUDY MODULE 32 and 33 FROM KAMAL SIR NOTES

IMP OBSERVATION

- Database-specific implementation classes are not in the java.sql package. The implementation classes are in database drivers.
- The DriverManager.getConnection() method can be called with just a URL. It is also overloaded to take the URL, username, and password.
- If preparedStatement query has only one bind variable "?" and code try to set more then one bind variable the code compiles but give an Exception at the runtime. on the second "set" line.
- If the preparedStatement query has two bind variables but the code sets only one and then execute that query an Exception will be thrown at the line of execution.
- DriverManager is a concrete class but not an implementation of Driver Interface.
- If a preparedStatement have two bind variable, we set both (1,2) and execute, then again we set 1 and execute, here no Exception is thrown because it reuse the 2nd bind variable from previous execute.
- If we use executeUpdate() instead if executeQuery() it will throw an Exception at the runtime.

JDBC Connection URL:


Subprotocol
(Product/Vendor name)
Protocol
Subname
(Database specific connection details)
jdbc:postgres://localhost:5432/zoo
Colon separators

`Class.forName()` loads a class before it is used. With newer drivers, `Class.forName()` is no longer required.

`Class.forName()` throws ClassNotFoundException (checked)

Always use an `if` statement or `while` loop when calling `rs.next()`.

Column indexes begin with 1.

Summary

There are four key SQL statements you should know for the exam, one for each of the CRUD operations: create (`INSERT`) a new row, read (`SELECT`) data, update (`UPDATE`) one or more rows, and delete (`DELETE`) one or more rows.

For the exam, you should be familiar with five JDBC interfaces: `Driver`, `Connection`, `PreparedStatement`, `CallableStatement`, and `ResultSet`. The interfaces are part of the Java API. A database specific JAR file provides the implementations.

To connect to a database, you need the JDBC URL. A JDBC URL has three parts separated by colons. The first part is `jdbc`. The second part is the name of the vendor/product. The third part varies by database, but it includes the location and/or name of the database. The location is either `localhost` or an IP address followed by an optional port.

The `DriverManager` class provides a factory method called `getConnection()` to get a `Connection` implementation. You create a `PreparedStatement` or `CallableStatement` using `prepareStatement()` and `prepareCall()`, respectively. A `PreparedStatement` is used when the SQL is specified in your application, and a `CallableStatement` is used when the SQL is in the database. A `PreparedStatement` allows you to set the values of bind variables. A `CallableStatement` also allows you to set `IN`, `OUT`, and `INOUT` parameters.

When running a `SELECT` SQL statement, the `executeQuery()` method returns a `ResultSet`. When running a `DELETE`, `INSERT`, or

`UPDATE` SQL statement, the `executeUpdate()` method returns the number of rows that were affected. There is also an `execute()` method that returns a `boolean` to indicate whether the statement was a query.

You call `rs.next()` from an `if` statement or `while` loop to advance the cursor position. To get data from a column, call a method like `getString(1)` or `getString("a")`. Column indexes begin with `1`, not `0`. In addition to getting a `String` or primitive, you can call `getObject()` to get any type.

It is important to close JDBC resources when finished with them to avoid leaking resources. Closing a `Connection` automatically closes the `Statement` and `ResultSet` objects. Closing a `Statement` automatically closes the `ResultSet` object. Also, running another SQL statement closes the previous `ResultSet` object from that `Statement`.

Exam Essentials

Name the core five JDBC interfaces that you need to know for the exam and where they are defined. The five key interfaces are `Driver`, `Connection`, `PreparedStatement`, `CallableStatement`, and `ResultSet`. The interfaces are part of the core Java APIs. The implementations are part of a database driver JAR file.

Identify correct and incorrect JDBC URLs. A JDBC URL starts with `jdbc:`, and it is followed by the vendor/product name. Next comes another colon and then a database - specific connection string. This database - specific string includes the location, such as `localhost` or an IP address with an optional port. It may also contain the name of the database.

Describe how to get a *`Connection`* using *`DriverManager`*. After including the driver JAR in the classpath, call `DriverManager.getConnection(url)` or `DriverManager.getConnection(url, username, password)` to get a driver - specific `Connection` implementation class.

Run queries using a *`PreparedStatement`*. When using a `PreparedStatement`, the SQL contains question marks (`?`) for the parameters or bind variables. This SQL is passed at the time the `PreparedStatement` is created, not when it is run. You must call a setter for each of these with the proper value before executing the query.

Run queries using a *`CallableStatement`*. When using a `CallableStatement`, the SQL looks like `{ call my_proc(?)}`. If you are returning a value, `{?= call my_proc(?)}` is also permitted. You must set any parameter values before executing the query. Additionally, you must call `registerOutParameter()` for any `OUT` or `INOUT` parameters.

Choose which method to run given a SQL statement. For a `SELECT` SQL statement, use `executeQuery()` or `execute()`. For other SQL statements, use `executeUpdate()` or `execute()`.

Loop through a *`ResultSet`*. Before trying to get data from a `ResultSet`, you call `rs.next()` inside an `if` statement or `while` loop. This ensures that the cursor is in a valid position. To get data from a column, call a method like `getString(1)` or `getString("a")`. Remember that column indexes begin with 1.

Identify when a resource should be closed. If you're closing all three resources, the `ResultSet` must be closed first, followed by the `PreparedStatement`, `CallableStatement`, and then followed by the `Connection`.

Chapter 22: Security

IMP OBSERVATIONS

- serialPersistentFields is used to mark field whitelist
- transient used to blacklist the field.
- Inclusion attacks occur when multiple files or components are embedded within a single entity, such as a zip bomb or the billion laughs attack. Both can be thwarted with depth limits.

- To make a class immutable:
  i) make all instance variable private
  ii) private variable should not have any setter methods.
  iii) either class should be marked final or must contain all final methods.
  iv) class can have public constructor, no issue
  v) Make a defensive copy of variable in the constructor before assing to the instance variable. (in case of List etc, in case of string not required coz it is immutable).

- To make class serializable
  i) implement java.io.Serializable Marker Interface

- To make class well Encapsulated
  i) Make all variable private.

Points to remember for Serialization

1. Only object of those class are Serialized which implement java.io.Serializable Interface. (It is a Marker Interface)
2. If parent class has implemented Serializable then subclass is not required to do so, but vice-versa is not acceptable.
3. Only non-static members are saved via Serialization process.
4. Static and transient members are not saved via Serialization process.
5. If you don't want to save value of a non-static members, mark it as transient.
6. Constructor of object is never called when an object is deserialized.
7. During Deserialization transient variable get default value(null/0/false)
8. During Deserialization static variable get the value that the class holds during Deserialization.

TABLE 22.2 Methods for serialization and deserialization

Return type	**Method**	**Parameters**	**Description**
`Object`	`writeReplace()`	None	Allows replacement of object *before* serialization
`void`	`writeObject()`	`ObjectInputStream`	Serializes optionally using `PutField`
`void`	`readObject()`	`ObjectOutputStream`	Deserializes optionally using `GetField`
`Object`	`readResolve()`	None	Allows replacement of object *after* deserialization

Summary

When designing a class, think about what it will be used for. This will allow you to choose the most restrictive access modifiers that meet your requirements. It will also help you determine whether subclasses are needed or whether the class should be `final`. If instances of the class are going to be passed around, it may make sense to make the class immutable so the state is guaranteed not to change.

Injection is an attack where dangerous input can run. SQL injection is prevented using a `PreparedStatement` with bind variables. Command injection is prevented with input validation and security policies. Whitelisting and the principle of least privilege provide the safest combination.

Confidential information must be handled carefully. It should be carefully dealt with in log files, output, and exception stack traces. Confidential information must also be protected in memory through the proper data structures and object lifecycle.

Object serialization and deserialization needs to be designed with security in mind as well. The `transient` modifier flags an instance variable as not being eligible for serialization. More granular control can be provided with the `serialPersistentFields` constant. It is used to constrain the `writeObject()` method with `PutField` and the `readObject()` method with `GetField`. Finally, the `readResolve()` and `writeReplace()` methods allow you to return a different object or class.

Regardless of whether you are using serialization, objects must take care that the constructor cannot call methods that subclasses can override. Methods that are called from the constructor should be `final`. Making the constructor `private` or the class `final` also meets this requirement.

Finally, applications must protect against denial of service attacks. The most fundamental technique is always using trywith - resources to close resources. Applications should also validate file sizes and input data to ensure data structures are used properly.

Exam Essentials

Identify ways of preventing a denial of service attack. Using a try - with - resources statement for all I/O and JDBC operations prevents resource leaks. Checking the file size when reading a file prevents it from using an unexpected amount of memory. Confirming large data structures are being used effectively can prevent a performance problem.

Protect confidential information in memory. Picking a data structure that minimizes exposure is important. The most common one is using `char[]` for passwords. Additionally, allowing confidential information to be garbage collected as soon as possible reduces the window of exposure.

Compare injection, inclusion, and input validation. SQL injection and command injection allow an attacker to run expected commands. Inclusion is when one file includes another. Input validation checks for valid or invalid characters from users.

Design secure objects. Secure objects limit the accessibility of instance variables and methods. They are deliberate about when subclasses are allowed. Often secure objects are immutable and validate any input parameters.

Write serialization and deserializaton code securely. The `transient` modifier signifies that an instance variable should not be serialized. Alternatively, `serialPersistenceFields` specifies what should be. The `readObject()`, `writeObject()`, `readResolve()`, and `writeReplace()` methods are optional methods that provide further control of the process.