Default initialization values by type (of static/class and instance variables)

Variable type boolean

byte, short, int, long

float, double char

All object references (everything else) String

- Default initialization value - false

- 0 (in the type's bit-length)

- 0.0 (in the type's bit-length)

- ‘\u0000' (NUL)

- null

- null

Primitive wrapper classes comparison

Equals method of a primitive wrapper class ( e.g. java.lang.Integer, Double, Float etc) are

1. symmetric => a.equals(b) returns same as b.equals(a)

2. transitive => if a.equals(b) and b.equals(c) return true, then a.equals(c) returns true.

3. reflexive => a.equals(a) return true.

Order of operator precedence

Operator -> Symbols and examples -> Associativity

Array access, property access, brackets -> [] . () -> left to right

Post-unary operator -> expression++, expression-- -> not associative

Pre-unary operator -> ++expression, --expression -> right to left

Other unary operators -> +, -, ! -> right to left

Multiplication/Division/Modulus -> \*, /, % -> left to right

Addition/Subtraction -> +, - -> left to right

Shift operators -> <<, >>, >>> -> left to right

Relational operators -><, >, <=, >=, instanceof -> not associative

Equal to/not equal to -> ==, != -> left to right

Logical operators -> &, ^, | -> left to right

Short-circuit logical operators -> &&, || -> left to right

Ternary operator -> bool expr ? expr1 : expr2 -> right to left

Assignment operators -> =, +=, -=, \*=, /=, %=, &=, -> right to left

>>=, >>>=, <<=, ^=, !=

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.

1. 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.
2. A floating-point primitive is considered a double if we omit the "f"/"F" after the primitive. We have to explicitly state with "F"/"f" after a primitive if we are declaring a float value.
3. We can cast a smaller type to a larger type without using an implicit cast. We need to use an explicit cast to casts from a larger type to a smaller type. This concerns both integral and decimal types.
4. We can cast an integral type to a floating-point type without explicit conversion.
5. We have to use explicit cast to cast a floating-point type to an integral type.

implicit narrowing is permitted only among byte, char, short, and int. (without explicit cast)

1. The "d"/"D" letter that denotes a double value can be omitted. Floating point types are by default double if not specified otherwise

Java unary operators

unary operator -> Description

+ -> 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

! -> Inverts a Boolean's logical value

Logical Operators:

* AND is only true if both operands are true.
* Inclusive OR is only false if both operands are false.
* Exclusive OR is only true if the operands are different.

Data types supported by switch statements include the following:

* int and Integer
* byte and Byte
* short and Short
* char and Character
* int and Integer
* String
* enum values

Note that boolean, float, double and long and their associated wrapper classes, are not supported by switch statements. Note: Although it is not required for the exam to know the integral values of characters, it is good to know that all English letters (upper case as well as lower case) as well as 0-9 are below 127 and so are assignable to byte. Char numeric values are between 0 and 65,535.

When we omit the break keyword, the execution of all subsequent case and default statements will occur until a case/default statement with "break" is executed.

There's no restriction for the position of the default statement. The default statement isn't mandatory.

|  |  |
| --- | --- |
| Binary search rules (binarySearch): Scenario  Target element found in sorted array Target element not found in sorted array  Unsorted array | — Result   * Index of match * Negative value showing one smaller than the negative of index, where a match needs to be inserted to preserve sorted order * this result isn't predictable |

Methods in LocalDate, LocalTime, LocalDateTime

Method name -> LocalDate -> LocalTime -> LocalDateTime

plusYears / minusYears Yes No Yes

plusMonths / minusMonths Yes No Yes

plusWeeks / minusWeeks Yes No Yes

plusDays / minusDays Yes No Yes

plusHours / minusHours No Yes Yes

plusMinutes / minusMinutes No Yes Yes

plusSeconds / minusSeconds No Yes Yes

plusNanos / minusNanos No Yes Yes

Looking at it a different way, the protected rules apply under two scenarios:

* A member is used without referring to a variable. This is the case on lines 5 and 6. In this case, we are taking advantage of inheritance and protected access is allowed.
* A member is used through a variable. This is the case on lines 10, 11, 15, and 16. In this case, the rules for the reference type of the variable are what matter. If it is a subclass, protected access is allowed. This works for references to the same class or a subclass.

Order Java uses to choose the right overloaded method

Rule

Exact match by type Larger primitive type Autoboxed type Varargs

* > Example of what will be chosen for glide(1,2) -> public String glide(int i, int j) {}

-> public String glide(long i, long j) { }

* > public String glide(Integer i, Integer j) {}
* > public String glide(int... nums)

Order of Initialization

1. If there is a superclass, initialize it first (we'll cover this rule in the next chapter. For now, just say "no superclass" and go on to the next rule.)
2. Static variable declarations and static initializers in the order they appear in the file.
3. Instance variable declarations and instance initializers in the order they appear in the file.
4. The constructor.

Rules for JavaBeans naming conventions Rule

Properties are private

* > Example
* > private int numEggs;

-> public boolean isHappy() { return happy; }

* > public int getNumEggs() { return numEggs; }

-> public void setHappy(boolean happy)

{ this.happy = happy; }

* > public void setNumEggs(int num) { numEggs = num; }

Getter methods begin with is if the property is a boolean.

Getter methods begin with get if the property is not a boolean.

Setter methods begin with set.

The method name must have a prefix of set / get / is, followed by the first letter of the property in uppercase, fol­lowed by the rest of the property name.

Access modifiers restrictiveness (from the most restrictive to the least):

1. private — only within the particular class
2. package private / default — only by classes from the same package
3. protected — package private + child classes can access it
4. public — all classes

Hiding Static Methods

First, the four previous rules for overriding a method must be followed when a method is hidden. In addition, a new rule is added for hiding a method, namely that the usage of the static keyword must be the same between parent and child classes. The following list summarizes the five rules for hiding a method:

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 or more accessible than the method in the parent class.
3. The method in the child class cannot throw a checked exception that is new or

not a descendant of the exception thrown in the parent class method.

1. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as co-variant return types.
2. The method defined in the child class must be marked as static if it is marked as

static in the parent class (method hiding). Likewise, the method must not be marked as static in the child class if it is not marked as static in the parent class (method

overriding).

Abstract Class Definition Rules:

1. Abstract classes cannot be instantiated directly.
2. Abstract classes may be defined with any number, including zero, of abstract and non-abstract methods.
3. Abstract classes cannot be marked as private or final.
4. An abstract class that extends another abstract class inherits all of its abstract methods as its own abstract methods.
5. The first concrete class that extends an abstract class must provide an implementation for all of the inherited abstract methods.

Abstract Method Definition Rules:

1. Abstract methods may only be defined in abstract classes.
2. Abstract methods cannot be declared private or final.
3. Abstract methods must not provide a method body/implementation in the abstract class for which is it declared.
4. Implementing an abstract method in a subclass follows the same rules for overriding a method. For example, the name and signature must be the same, and the visibility of the method in the subclass must be at least as accessible as the method in the parent class.

Hiding private methods:

There's a catch when hiding private methods. The hidden method can still be called. This depends on the type of the reference to the object and the class context in which they are called (Test 5 from the book, question 20).

Inheriting an Interface

There are two inheritance rules you should keep in mind when extending an interface:

1. An interface that extends another interface, as well as an abstract class that implements an interface, inherits all of the abstract methods as its own abstract methods.
2. The first concrete class that implements an interface, or extends an abstract class that implements an interface, must provide an implementation for all of the inherited abstract methods.

Interface Variables

Like interface methods, interface variables are assumed to

be public. Unlike interface methods, though, interface variables are also assumed to be static and final.

Here are two interface variables rules:

1. Interface variables are assumed to be public, static, and final. Therefore, marking a variable as private or protected will trigger a compiler error, as will marking any variable as abstract.
2. The value of an interface variable must be set when it is declared since it is marked as final.

Default interface method rules:

1. A default method may only be declared within an interface and not within a class or abstract class.
2. A default method must be marked with the default keyword. If a method is marked as default, it must provide a method body.
3. A default method is not assumed to be static, final, or abstract, as it may be used or overridden by a class that implements the interface.

Static interface method rules:

1. Like all methods in an interface, a static method is assumed to be public and will not compile if marked as private or protected.
2. To reference the static method, a reference to the name of the interface must be used.
3. It is illegal to invoke a static method of an interface using a reference variable. If you want to invoke a static method of an interface, you need to use the name of the interface.

We can summarize this principle with the following two rules:

1. The type of the object determines which properties exist within the object in memory.
2. The type of the reference to the object determines which methods and variables are accessible to the Java program.

Here are some basic rules to keep in mind when casting variables:

1. If a class shares an IS-A or inheritance relationship with another class or interface their variables can be cast to each other's types.
2. Casting an object from a child to a parent doesn't require an explicit cast.
3. Casting an object from a parent to a child requires an explicit cast.
4. The compiler will not allow casts to unrelated types.
5. Even when the code compiles without issue, an exception may be thrown at runtime if the object being cast is not actually an instance of that class.
6. Reference variables only refer to an object but don't contain the object itself. Casting a reference variable doesn't touch the object it refers to, but only labels this object in another way, expanding or narrowing opportunities to work with it. Upcasting narrows the list of methods and properties available to this object and downcasting can extend it.
7. There's a difference between the type of a reference and the type of an object.
8. Variables can only be hidden, not overriden. This is regardless of the access modifier.
9. Instance variables can be hidden by local variables within a method if they have the same name. In such a case we need to use this.varName explicitly to refer to the instance variable.

Types of exceptions

|  |  |  |  |
| --- | --- | --- | --- |
| TYPe | -> How to recognize | -> Okay to catch? -> required to handle or  declare? | |
| Runtime exception | -> Subclass of Runtime Exception | -> Yes | -> No |
| Checked exception | -> Subclass of Exception but not of | -> Yes | -> Yes |
|  | Runtime exception |  |  |
| Error | -> Subclass of Error | -> No (but can be catched) | -> No |

Runtime Exceptions

Runtime exceptions extend RuntimeException. They don't have to be handled or declared. They can be thrown by the programmer or by the JVM. Common runtime exceptions include the following:

**ArithmeticException -** Thrown by the JVM when code attempts to divide by zero **ArraylndexOutOfBoundsException -** Thrown by the JVM when code uses an illegal index to access an array

**ClassCastException** Thrown by the JVM when an attempt is made to cast an excep­tion to a subclass of which it is not an instance

**IllegalArgumentException** Thrown by the programmer to indicate that a method has been passed an illegal or inappropriate argument

**NullPointerException** Thrown by the JVM when there is a null reference where an object is required

**NumberFormatException -** Thrown by the programmer or JVM when an attempt is made to con­vert a string to a numeric type but the string doesn't have an appropriate format. E.g. when we use the parseXXX methods from the primitive wrapper classes with bad input. **StringlndexOutOfBoundsException —** Thrown by the JVM when an attempt is made to access a character in a string that isn't set. (e.g. with the charAt method).

**IndexOutOfBoundsException —** parent exception of the previous two. Thrown by the JVM when an attempt is made to access a position in a collection that is not yet set.

**DateTimeException —** Parent exception of the ones thrown to indicate a problem while working date/time objects.

**NumberFormatException**  - thrown by the program when an attempt is made to convert a string to a numeric type, but the string doesn’t have an appropriate type.

**IllegalStateException –** attempting to run an invalid operation in collections and concurrency

**UnsuportedOperationException -** attempting to run an invalid operation in collections and concurrency

**java.lang.ArrayStoreException**  - trying to store the wrong data type in an array

**java.time.DateTimeException** – Receiving an invalid format string for a date

**java.util.MissingResourceException –** trying to access a key or resource bundle that does not exist

**ConcurrentModificationException –** when two threads try to modify the same non-concurrent collection at the same time

Errors — they extend the error class.

They are thrown by the JVM and can be handled by the developer but this is not good coding practice: **ExceptioninInitializerError** Thrown by the JVM when a static initializer throws

an exception and doesn't handle it

**StackOverfiowError** Thrown by the JVM when a method calls itself too many times (this is called infinite recursion because the method typically calls itself without end)

**NoClassDefFoundError** Thrown by the JVM when a class is available at compile time but not at runtime.

**AssertionError —** Thrown by the JVM when an assertion statement has failed

Constructor Definition Rules:

1. The first statement of every constructor is a call to another constructor within the class using this() , or a call to a constructor in the direct parent class using super() .
2. The super() call cannot be used after the first statement of the constructor.
3. If no super() call is declared in a constructor, Java will insert a no-argument super() as the first statement of the constructor.
4. If the parent doesn't have a no-argument constructor and the child doesn't define any constructors, the compiler will throw an error and try to insert a default no-argument constructor into the child class.
5. If the parent doesn't have a no-argument constructor, the compiler requires an explicit call to a parent constructor in each child constructor.
6. A constructor cannot be final.

The compiler performs the following checks when you override a non-private method:

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 or more accessible than the method in the parent class.
3. The method in the child class can only throw the same exception as the over-ridden method or its subclass.
4. If the method returns a value, it must be the same or a subclass of the method in the parent class, known as **co-variant** return types.

Defining an Interface

It may be helpful to think of an interface as a specialized kind of abstract class, since it shares many of the same properties and rules as an abstract class. The following is a list of rules for creating an interface, many of which you should recognize as adaptions of the rules for defining abstract classes.

1. Interfaces cannot be instantiated directly.
2. An interface is not required to have any methods.
3. An interface cannot be marked as final.
4. All top-level interfaces are assumed to have public or default access, and they must include the abstract modifier in their definition. Therefore, marking an interface as private, protected, or final will trigger a compiler error, since this is incompatible with these assumptions.
5. All non-default methods in an interface are assumed to have the modifiers abstract and public in their definition. Therefore, marking a method as private, protected, or final will trigger compiler errors as these are incompatible with the abstract and public keywords.

ofLocalized methods

DateTimeFormatter

f = DateTime

Formatter.

(FormatStyle.SHORT); Calling f.format (localDate) Calling f.format (localDateTime) Calling

f.format (localTime)

|  |  |
| --- | --- |
| ofL ocalizedDate Legal - shows whole object  ofLocalizedDateTime Throws runtime exception ofL ocalizedTime Throws runtime exception | Legal - shows just date part Throws runtime  exception  Legal - shows whole object Throws runtime  exception  Legal - shows just time part Legal - shows  whole object  Check page 271 on the OCP guide for how those work with ZonedDateTime |

Converting value from a String to primitive or wrapper class.

Wrapper class Converting **String**  to **primitive**

Boolean Boolean.parseBoolean("true");

Byte Byte.parseByte("1");

Short Short.parseShort("1");

Integer Integer.parseInt("1");

Long Long.parseLong("1");

Float Float.parseFloat("1");

Double Double.parseDouble("1");

Character None

Converting **String** to **wrapper object** (static methods)

Boolean.valueOf("TRUE"); Byte. valueOf ("2"); Short.valueOf("2"); Integer.valueOf("2"); Long.valueOf("2"); Float.valueOf("2.2"); Double.valueOf("2.2"); None

Converting **wrapper object**  to **primitive** (instance methods)

(new Boolean("true")).booleanValue();

(new Byte(0)).byteValue(); (new Short(123)).shortValue(); (new Integer(123)).intValue(); (new Long(123)).longValue();

(new Float("123.0f")).longValue(); (new Double(123.0)).doubleValue(); (new CharacterCc'D.charValue();

All of those wrapper classes are immutable - a given instance always has the same value inside of it.

OCP Checked exceptions:

**java.text.ParseException -** converting a String to a number

**java.io.IOException –** parent class of I/O related exceptions; all child classes are checked exceptions

**java.io.FileNotFoundException –** child class of IOException

**java.io.NotSerializableException –** child class of IOException

**java.sql.SQLException** - dealing with database issues. Checked exception, as well as all its child classes.

**java.concurrent.InterruptedException –** Thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity.

**java.util.concurrent.ExecutionException –** exception thrown when attempting to retrieve the result of a task that aborted by throwing an exception. This exception can be inspected using the getCause() method.

**java.util.concurrent.BrokenBarrierException -** Exception thrown when a thread tries to wait upon a barrier that is in a broken state, or which enters the broken state while the thread is waiting

**java.io.FileNotFoundException –** thrown by FileInputStream and FileOutputStream

**java.io.NotSerializableException** – thrown when we try to deserialize an object which doesn’t implement Serializable. We can use the transient keyword to avoid this exception, but the data we attempt ed to serialize will be lost. Static class members are ignored during serialization and deserialization process.

**java.net.URISyntaxException -**  thrown when a string cannot be parsed as a valid URI

**java.nio.file.AtomicMoveNotSupportedException -**  thrown when a file cannot be moved as an atomic file system operation

**java.nio.file.FileSystemLoopException –** thrown by the NIO.2 classes when a directory/symlink cycle is detected.

Arrays

Valid ways to initialize an array:

Ma] myArray = new int[3];

int[] myArray = {1, 2, 3};

int[] myArray = new int[]11, 2, 31

this is the way to cast an array after you've already declared it: String[] myStringArray;

myStringArray = new Stringll { "a", "b", "c"}

Multidimensional array: int[][] num = new int[4][5]; int num[][] = new int[4][5]; int[] numll = new int[4][5];

Ragged array (non-rectangular array):

int[][] num = new int[5][]; //this will instantiate only the first dimension of the array

num[0] = new int[1]; num[1] = new int[5]; num[2] = new int[2]; num[3] = new int[3]; num[4] = new int[4];

Ragged array:

int[][] num = {1}, {1,2}, {1, 2, 3, 4, 5}, {1, 2}, 11, 2, 31 }

**In** Java, arrays are just like regular Objects and arrays of different types have different class names. For example, the class name of an int[] is [I and the class name for int[][] is [[I. For array classes, the isArray() method of a Class returns true. For example, twoD.getClass().isArray() will return true.

There is a subtle difference between: int[] i; and int i[]; although in both the cases, i is an array of integers. The difference is if you declare multiple variables in the same statement such as: int[] i,, , j; and int i[], j - j is not of the same type in the two cases.

You can't specify array length if you are initializing it at the same place. For example the following statement will not compile:

new Object[1]{ new Object() };

It is fully possible to instantiate arrays with 0 elements. When we try to access any of their elements an ArraylndexOutOfBounds Exception is thrown. (reference: the syntax for Collections.asArray to convert an ArrayList to an array).

In java arrays are similar to objects. The have a length property which contains the size of the array. Their default values are initialized respectively to the type of values they contain. They have a toString method. They are a reference type.

Exception handling rules:

* 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. This happens when a superclass is caught before a subclass.

- A try block cannot exist without either a finally or catch block (excluding the try-with-resources syntax for AutoCloseable objects).

- we can have multiple catch statements

* we can't have a try/catch/finally statement without the curly braces even if only a single statement is involved

- we cannot place the finally statement before the catch statement

* when we have multiple catch statements with related exceptions the child exceptions must be handled first, otherwise the code will not compile.
* for the sake of the exam: you can declare a method as Exception as the return type
* for the sake of the exam: you can declare a method to throw Throwable and Error
* a method can be declared to throw unchecked exceptions, independent of the exceptions thrown by the method it overrides
* the exact same exception type cannot be specified in two different catch blocks
* a compile-time error will be raised if we are handling an exception that isn't being thrown within the try statement or the methods called within the try statement or they aren't declared to throw it - an uncaught declared exception stops execution
* Any checked exceptions must either be handled using a try block or the method that generates the exception must declare that it throws that exception.

- Compiler warnings (not errors) tell you something is wrong or suspicious without failing compilation. - within a method declaring to throw an exception can throw a child exception of the one declared. A parent exception cannot be thrown.

- Since Java 7 we can collapse the catch blocks into a single one:

try { //logic that throws any of the three exceptions

} catch (SQL Exception | IOException | RuntimeException e)

/\* //In this block, the class of the actual exception object will be whatever exception is

thrown at runtime.

//But the class of the reference e will be the closest common super class of all the exceptions in the catch block.

//In this case, it will be java.lang.Exception because that is the most specific class that is a super class for all the three exceptions.

//The exceptions here cannot be related by subclassing. A compiler error will be thrown if so \*/

}

- we cannot access variables declared in the try block within the catch block

- the finally block can be omitted in a try-with-resources syntax, but it is implicitly added by the JVM

- the resources created within the try block in try-with-resources syntax are only available within the scope of the try block

- only classes that implement the AutoCloseable interface can be passed to a try-with-resources statement. Closeable was added first and then it was changed to extend AutoCloseable

- we can add a catch block in a try-with-resources syntax in order to catch an exception thrown by the close() method implementation. If the close() method implementation throws a checked exception, there should be a catch statement which handles it.

- in case of multiple exceptions within the try block when using the try-with-resources syntax, all but the first are handled (the one thrown in the try(…) {} expression. The rest are called **suppressed exceptions**. They can be accessed via the getSuppressed() method of the Exception object within the catch block

- suppressed exceptions need not be handled. Java *remembers* the suppressed exceptions that go with a primary exception even if we don’t handle them in the code.

- close() in the AutoCloseable interface has Exception in the signature and close() in the Closeable interface has IOException in the signature

Order of operator precedence (extended)

**Level — Operator — Description Associativity**

16 [] access array element left to right

. access object member

()

15 val++ unary post-increment not associative

val- - unary post-decrement

14 ++val unary pre-increment right to left

- -val unary pre-decrement

+ unary plus

unary minus

! unary logical NOT

unary bitwise NOT

13 0 cast right to left

new object creation

12 \* / % multiplication, division, left to right

modulus(remainder)

11 + - addition, substraction left to right

+ string concatenation

10 <<, >> shift operators left to right

>>>

9 <<=, relational not associative

>>=, instanceof

8 == equals left to right

!= not equals

7 & bitwise AND left to right

6 ^ bitwise XOR left to right

5 | bitwise OR left to right

4 && logical AND left to right

3 || | logical OR left to right

2 ?: ternary right to left

1 = += -= &= ^=

\*= /= %= |= <<= >>= >>>= assignment right to left

Declaration/definition/initialization rules:

1. Chaining to use a value of a variable at the time of declaration is not allowed.

For example, this isn't allowed:

int a = b = c = 100;

Had b and c been defined previously, this would compile.

1. Final fields can be hidden.
2. Final variables must be explicitly initialized.
3. Static fields can be left uninitialized and they will receive a default value.
4. We can use null references to access static members and methods (test 7, question 4).
5. We can declare local variables as final also. In case we use it with reference variables the final keyword modifies the objects it points to. It cannot reference another object onwards. The object's itself can be mutated accordingly to their access modifiers.
6. Class fields can are hidden, they aren't overriden when fields with the same identifier exist in their child class.
7. for the sake of the exam: we can use final for local variables, but not static.
8. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from final array or final collection.

Interfaces

Interfaces can contain abstract, default and static methods. All abstract, default and static methods in an interface are implicitly public. An interface can contain constant declarations. All constant values defined in an interface are implicitly public, static and final. The abstract keyword can be omitted when defining abstract methods within interfaces. The definition needs to be omitted to be declared implicitly abstract. An interface an extend other interfaces but it cannot implement them.

When you extend an interface that contains a default method you can:

- redefine the default method which overrides it

- re-declare the default method as abstract

- inherit the default method by not mentioning it at all

When a class implements more than one interface and in some of them there are constants with the same name a compile time error will be thrown because of ambiguous field names.

byte - 1 byte (8 bits) - -2A7 to 2A7-1— from -128 to 127

short - 2 bytes (16 bits)- -2A15 to 2A15-1— from -32,768 to 32,767

int - 4 bytes (32 bits) - -2A31 to 2A31-1— from -2147483648 to 2147483647

long - 8 bytes (64 bits) - -2A63 to 2A63-1 — from -9223372036854775808 to 9223372036854775807 char — 16 byte UTF-16 character — unsigned type - Au0000' (or 0) to ‘\uffff' (or 65,535)

Abstract classes

An abstract class is a class that is declared abstract — it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed. An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

*abstract void move To (double deltaX, double deltaY);*

if a class includes abstract methods, then the class itself must be declared abstract. The abstract keyword cannot be added to methods within concrete classes.

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, it is does not, then the subclass must also be declared abstract.

Defining a Method with the Same Signature as a Superclass's Method

**Superclass Instance Method Superclass Static Method Subclass Instance Method** Overrides Generates a compile-time error   
**Subclass Static Method** Generates a compile-time error Hides

Abstract classes and interfaces comparison:

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without implementation. **However, with abstract classes, you can declare fields that are not static and final, and define public, protected and private concrete methods. With interfaces, all fields are automatically public, static and final, and all methods that you declare or define (as default methods) are public.** In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces. An abstract class can have static fields and static methods. You can use these static members with a class reference (for example, AbstractClass.staticMethod()) as you would with a concrete class.

When an abstract class implements an interface it may not implements all the methods in the interface. Consider using abstract classes if any of these statements apply to your situation:

* you want to share code among several closely related classes
* you expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).

- you want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

Consider using interfaces if any of these statements apply to your situation:

* you expect that unrelated classes would implement your interface.
* you want to specify the behavior of a particular data type, but not concerned about who implements this behavior

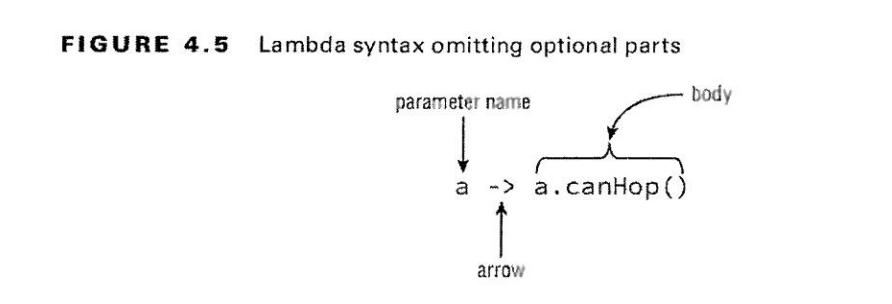
- you want to take advantage of multiple inheritance of type

Lambda syntax

Lambdas can access variables from the outer scope. They can access instance and static variables and mutate them. Method parameters and local variables can also be accessed if they are not mutated.

Lambdas work with interfaces that have only one method. These are called functional interfaces —interfaces that work with functional programming.

In order to declare a method which accepts a lambda as its parameter we need to place Predicate<Typename> as its parameter.

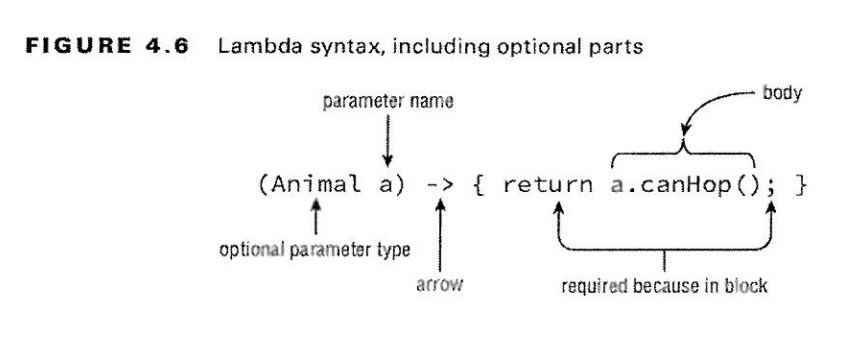


The parentheses can only be omitted if there is a single parameter and its type is not

explicitly stated. We can omit braces when we only have a single statement. We did this with if statements and loops already. What is different here is that the rules change when you omit the braces. Java doesn't require you to type return or use a semicolon when no braces are used. This special shortcut doesn't work when we have two or more statements. Remember that the parentheses are only optional when there is one parameter and it doesn't have a type declared. When the lambda expression returns void the parentheses for the return type should be empty. Remember that the parameter list part of a lambda expression declares new variables that are used in the body part of that lambda expression. However, a lambda expression does not create a new scope for variables. Therefore, you cannot reuse the local variable names that have already been used in the enclosing method to declare the variables in you lambda expression. It would be like declaring the same variable twice:

— System.out.println("running...")

{ System. out.println("running..."); return; }



Polymorphism

There are two types of polymorphism:

1. Runtime polymorphism (overriding of functions) — in this type the call to an overriden function is resolved at runtime. Also known as dynamic polymorphism
2. Compile time polymorphism (overloading of functions) — also known as static polymorphism
3. Reference variables only refer to an object but doesn't contain the object itself. Casting a reference variable doesn't touch the object it refers to, but only labels this object in another way, expanding or narrowing opportunities to work with it. Upcasting narrows the list of methods and properties available to this object and downcasting can extend it.
4. The most important feature of polymorphism is to support *virtual* methods. A virtual method is a method in which the specific implementation is not determined until runtine. In fact, all non-final, non-static and non-private Java methods are considered virtual methods, since any of them can be overriden at runtime. What makes a virtual method special in Java is that if you call a method on an object that overrides a method, you get the overriden method, even if the call to the method is on a parent reference or within the parent class.
5. The default property values and access modifiers are determined by the reference type. For example:

A a = new A(); A b = new BO; - the visibility and default values of the properties in b will be those of A. While accessing a method or variable, the compiler will only allow you to access a method or variable that is visible through the class of the reference.

import java.nio.\*; 1/ NO GOOD — a wildcard only matches

//class names, not "file.\*Files"

import java.nio.\*.\*; // NO GOOD — you can only have one wildcard

//and it must be at the end

import java.nio.files.Paths.\*; /1 NO GOOD — you cannot import methods

//only class names

**NB!!!** importing by class name takes precedence over wildcards.

You can write either import java.util.\*; or import java.util.Date;. The tricky cases come about when other imports are present:

import java.util.\*;

import java.sql.\*; // DOES NOT COMPILE

When the class is found in multiple packages, Java gives you the compiler error:

The type Date is ambiguous

Multiple inheritance

1. Multiple inheritance by state — the ability to inherit fields from multiple classes
2. Multiple inheritance by type — the ability of a class to implement more than one class. In java this is the equivalent of when a class implements multiple methods.
3. Multiple inheritance by implementation — the ability to inherit method definitions from other classes. Default methods in interfaces introduce a form of multiple inheritance of implementation

**Dates, Strings and Localization**

Rules for properties files syntax:

1. Spaces at the beginning of a line are ignored
2. Spaces at the end of the line are not ignored
3. End a line with a backslash if you want to break the line for readability
4. You can use normal Java escape characters like \t and \n
5. Comments start with ! or #
6. We can use “:” as a delimiter also
7. We can use “ “ as a delimiter also. If we don’t use quotes for the value and it contains spaces the first space becomes the delimiter.
8. Spaces before or after the delimiter is ignored.
9. Properties files which start with the same prefix, delimited by “\_” are treated to be in the same resource bundle.
10. To implement a resource bundle in the Java source code you need to create a class which extends the ListResourceBundle abstract class. There are two main advantages of implementing a java class for a resource bundle instead of using a properties file: you can use a type that is not a string and you can create the values of the properties at runtime.
11. On the exam there are two methods for getting a resource bundle:

ResourceBundle.getBundle(“name”) – uses the default locale

ResourceBundle.getBundle(“name”, locale)

The exam either tells you what to assume as the default locale or uses the second approach.

1. Lookup page 264 on the study guide for the priority in which Java determines from where to read the locale data for a resource bundle.
2. The convention for storing variables in properties files is {0}. We java.text.MessageFormat can be used to substitute the variable in Java source code.

* **Exceptions and Assertions**

Possible outcomes when using native Java assertions:

* If assertions are disabled, Java skips the assertion and goes on with the code
* If they are enabled and the expression returns true nothing happens; the program continues execution
* If they are enabled and the bool expression is false the program stops execution by throwing a java.lang.AssertionError

Assertion syntax:

assert boolean\_expression;

assert Boolean\_expression: error\_message;

Java assertions aren’t enabled by default. We need to pass -enableassertions or -ea to the java runtime environment when we run CLI programs. We can enable or disable assertions for a specified class. We can use -da or -disableassertions flag to disable them.

Expressions tested in assertions can be surrounded by parentheses but this is not required.

* **Concurrency**

Reasons to prefer extending Thread instead of implementing Runnable or vice versa:

* If you need to define your own Thread rules upon which multiple tasks will rely, such as a priority Thread
* Since Java doesn’t support multiple inheritance, extending Thread does not allow you to extend any other class while implementing Runnable does
* Implementing Runnable is ofthen a better object-oriented design practice since it separates the task being performed from the Thread object performing it
* Implementing Runnable allows the class to be used by numerous Concurrency API classes

The java.concurrent.Executor service life cycle:

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated

Basic Concurrency / multi-threading glossary:

* Polling – the process of intermittently checking data at some fixed interval
* Shared environment – the threads in the same process share the same memory space and can communicate directly with one another
* Single-threaded process – a process which contains one thread
* Multi-threaded process – a process which contains more than one thread. All java applications are multi-threaded.
* A system-thread is a thread started by the JVM which runs in the background of the application.

For example, the garbage-collection thread is a system thread that is created by the JVM helping to free memory that is no longer in use. The execution of system threads most of the time is invisible to the application developer. System threads throw Errors instead of Exceptions.

* A user-defined thread is one defined by the application developer.
* A daemon thread is one which will not prevent the JVM from exiting when the program finishes. A Java program terminats when the only threads which are running are daemon threads. For example, if the garbage collection thread is the only one running left, the JVM will automatically shut down. Both system and user defined threads can be marked as daemon threads.
* Concurrency is the property of executing multiple processes and threads at the same time is referred to as concurrency.
* Context switch is the process of storing a thread’s current state and later restoring the state of the thread to continue execution.
* A thread priority is a numeric value associated with a thread that is taken into consideration by the thread scheduler when determining which threads should currently be executing. A thread can interrupt or supersede another thread.
* Thread-safety is the property of an object that guarantees safe execution by multiple threads at the same time.
* Atomic is the property of an operation to be carried out as a single unit of execution without any interference by another thread.
* A monitor or a lock is a structure that supports mutual exclusion (mutex) or the property that at most one thread is executing a particular segment of code at a given time. In Java we can specify a block of code with the synchronized keyword that can be executed only by a single thread at a time.
* A memory consistency error occurs when two threads have inconsistent views of what should be the same data.
* A parallel stream is a stream that is capable of processing results concurrently, using multiple threads.
* A serial stream is a stream in which the results are ordered, with only one entry being processed at a time.
* A stateful operation within a stream is an operation that can cause a side-effect from a lamba expression. A stateful lambda expression is one whose result depends on any state that might change during the execution of a pipeline.
* Java streams are ordered by default. We can create an unordered stream from an ordered one via the unordered method.
* Liveness – the ability of an application to execute in a timely manner. Liveness problems are those in which the application becomes unresponsive or in some kind of “stuck” state. There are three types of liveness issues within the exam: deadlock, starvation and livelock.
* Deadlock occurs when two or more threads are blocked forever, each waiting on the other
* Starvation occurs when a single thread is perpetually denied access to a shared resource or lock. The thread is still active, but it is unable to complete its work as a result of other threads constantly taking the resource that they are trying to access
* Livelock occurs when two or more threads are conceptually blocked forever, although they are each still active and trying to complete their task. Livelock is a special case of resource starvation in which two or more threads actively try to acquire a set of locks, are unable to do so and restart part of the process. Livelock mostly occurs as the result of two threads trying to resolve a deadlock
* A race condition occurs when two tasks, which should be completed sequentially, are completed at the same time

A screenshot of a social media post

Description automatically generated

A screenshot of a social media post

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

Description automatically generated

* **IO and NIO.2**

The java.io API defines two sets of classes for reading and writing streams – those with stream in their name and those with Reader/Writer in their name. Reader/Writer classes do not contain Stream in their name but they are in fact also a type of stream. Character stream classes are something referred to as convenience classes. Difference between streams and readers/writers:

* The stream classes are used for inputting and outputting all types of binary or byte data
* The reader and writer classes are used for inputting and outputting only characters and String data
* Most Reader classes have a corresponding Writer class. There are exceptions to this rule. For example, PrintWriter has no accompanying PrintReader class.

Difference between high-level and low-level streams:

* Low-level streams connect directory with the source of the data, such as a file, array or a String.
* High-level streams are built on top of another stream using wrapping. Wrapping is the process by which an instance is passed to the constructor of another class and operations on the resulting instance are filtered and applied to the original instance. The constructors of high-level streams often take a reference to the abstract parent class. For example, BufferedWriter takes a Writer class as argument to it constructors.

The parents of all stream classes:

* InputStream
* OutputStream
* Reader
* Writer

For convenience, the authors of the Java API include the name of the abstract parent class as the suffix of the child class. PrintStream, which is an OutputStream doesn’t follow this naming convention.

Rules for the java.io classes:

* A class with the word InputStream or OutputStream in its name is used for reading or writing binary data, respectively
* A class with the word Reader or Writer in its name is used for reading or writing character or string data, respectively
* Most but not all input classes have a corresponding output class
* A low-level stream connects directly with the source of the data
* A high-level stream is built on top of another stream using wrapping
* A class with Buffered in its name reads or writes data in groups of bytes or characters and often improves performance in sequential file system
* When wrapping a stream you can mix and match only types that inherit from the same abstract parent stream
* Streams are considered resources, it is imperative that they are closed. Otherwise, resource leaks may occur
* FileInputStream and FileOutputStream are the most basic streams for reading and writing bytes to a file. Their constructors take up a File object or a String representing a path to a file
* To serialize and deserialize objects with ObjectInputStream and ObjectOutputStream they need to implement the Serializable interface. It is a marker interface for the JVM, thus it has no methods. Any object references contained within a Serializable class need also to be Serializable. Object members marked as “transient” and “static” are avoided during serialization/deserialization. Adding an int property called serialVersionID is good practise. This is a special type of variable the JVM uses to identify a version of the class.

Try-with-resources rules:

* Resources are closed after the try clause ends and before any catch/finally clauses
* Resources are closed in the reverse order from which they were created

A picture containing screenshot

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**JDBC**

Name the core four JDBC interfaces that you need to know for the exam and where they

are defined. The four key interfaces are Driver, Connection, Statement, and ResultSet.

The interfaces are part of the core Java APIs. The implementations are part of a database client 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 also contains 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.

Create a Statement using different options. When creating a Statement, you can use

the defaults. Alternatively, you can specify the ResultSet type followed by the ResultSet

concurrency mode. The options for ResultSet type are TYPE\_FORWARD\_ONLY, TYPE\_SCROLL\_INSENSITIVE, and TYPE\_SCROLL\_SENSITIVE. The options for ResultSet concurrency mode are CONCUR\_READ\_ONLY and CONCUR\_UPDATABLE.

Choose which method on Statement 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 forward only 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.

Navigate within a scrollable ResultSet. The rows in a ResultSet are numbered starting

with 1. Calling absolute(4) moves the cursor to the fourth row. Calling absolute(0)

moves the cursor to a location immediately before the result. Calling absolute(-1) moves

the cursor to the last row.

Identify when a resource should be closed. If you’re closing all three resources, the

ResultSet must be closed first, followed by the Statement, and then followed by the

Connection. Closing an object later in this list automatically closes those earlier in the

list.

* **Functional programming**

Functional programming syntax / Java 8 Functional glossary and terms:

* The first parameter to the reduce() method is called the *identity*, the second is called the *accumulator* and the third parameter is called *combiner*. Check page 374 on the OCP guide.

Advanced class design

Design Patterns and principles

Generics and Collections

+ Functional Programming

+ Dates, Strings and Localization

+ Exceptions and Assertions

+ Concurrency

+ IO

+ NIO.2

+ JDBC