* The instance initializer blocks are executed after an implicit or explicit call to the parent classes constructor:

class Instrument {

Instrument() {

System.out.println("Instrument:constructor");

}

}

class Pencil extends Instrument {

public Pencil() {

System.out.println("Pencil:constructor");

}

{

System.out.println("Pencil:instance initializer");

}

public static void main(String[] args) {

new Pencil();

}

}

The output of the preceding code is

Instrument:constructor

Pencil:instance initializer

Pencil:constructor

RETURN TYPE:

Methods can’t be defined as overloaded methods if they only differ in their return types, as follows:

class Result {

**double** calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2;

}

**int** calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2;

}

}

The methods defined in the preceding code aren’t correctly overloaded methods—they won’t compile.

ACCESS MODIFIER

Methods can’t be defined as overloaded methods if they only differ in their access modifiers, as follows:

class Result {

**public** double calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2;

}

**protected** double calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2;

}

}

NONACCESS MODIFIER

Methods can’t be defined as overloaded methods if they only differ in their non-access modifiers, as follows:

class Result {

public **synchronized** double calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2;

}

public **final** double calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2;

}

}

EXAM TIP

If a parent or child class defines static initializer block(s), the execution start from parent then child class static initializer block.

ACCESS MODIFIERS

A derived class can assign the same or more access but not a weaker access to the overriding

method in the derived class:

class Book {

**protected** void review(int id, List names) {}

}

class CourseBook extends Book {

void review(int id, List names) {}//wrong we shouldn’t decrease the scope of access modifiers

**public** void review(int id, List names) {}//correct

}

**Won’t compile; overriding methods in derived classes can’t use a weaker access.**

NONACCESS MODIFIERS

A derived class can’t override a base class method marked as final:

class Book {

final void review(int id, List names) {}

}

class CourseBook extends Book {

void review(int id, List names) {}

}

**Won’t compile; final methods can’t be overridden.**

ARGUMENT LIST AND COVARIANT RETURN TYPES

When the overriding method returns a subclass of the return type of the overridden method, it’s known as a covariant return type. To override a method, the parameter list of the methods in the base and derived classes must be exactly the same. It you try to use covariant types in the argument list, you’ll end up overloading the methods and not overriding them.

For example

class Book {

void review(int id, **List names**) throws Exception {

System.out.println("Base:review");

}

}

class CourseBook extends Book {

void review(int id, **ArrayList names**) throws IOException {

System.out.println("Derived:review");

}

}

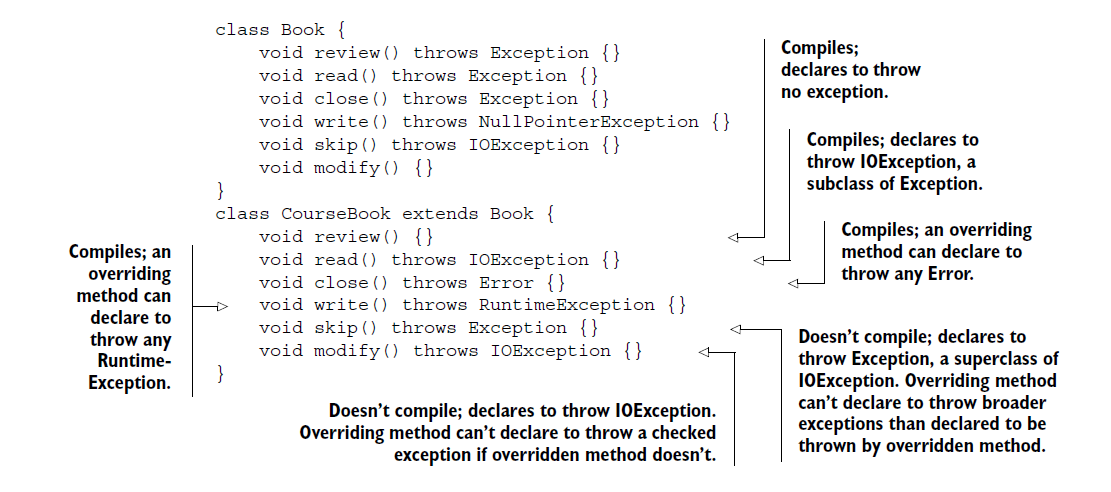
**Not overriding its overloading.**

EXCEPTIONS THROWN

An overriding method must either declare to throw no exception, the same exception, or a subtype of the exception declared in the base class method, or else it will fail to compile. This rule, however, doesn’t apply to error classes or runtime exceptions. We can throw any error or runtime exceptions.

EXAM TIP

An overriding method can declare to throw any Runtime-Exception or Error, even if the overridden method doesn’t.



CAN YOU OVERRIDE ALL METHODS FROM THE BASE CLASS OR INVOKE THEM VIRTUALLY?

The simple answer is no. You can override only the following methods from the base class:

* Methods accessible to a derived class
* Non static base class methods

METHODS ACCESSIBLE TO A BASE CLASS

The accessibility of a method in a derived class depends on its access modifier. For example, a private method defined in a base class isn’t available to any of its derived classes. **Also,** **a method with default access in a base class isn’t available to a derived class in another package.** A class can’t override the methods that it can’t access.

ONLY NONSTATIC METHODS CAN BE OVERRIDDEN

If a derived class defines a static method with the same name and signature as the one defined in its base class, it hides its base class method and doesn’t override it. You can’t override static methods. For example

class Book {

static void printName() {

System.out.println("Book");

}

}

class CourseBook extends Book {

static void printName() {

System.out.println("CourseBook");

}

}

Method printName() in class CourseBook hides printName() in class Book. It doesn’t override it. Because the static methods are bound at compile time, the method print-Name() that’s called depends on the type of the reference variable:

class BookExampleStaticMethod {

public static void main(String[] args) {

Book base = new Book();

base.printName();

Book derived = new CourseBook();

derived.printName();

}

}

*IDENTIFYING METHOD OVERRIDING, OVERLOADING, AND HIDING*

When a class extends another class, it can overload, override, or hide its base class methods. A class can’t override or hide its own methods—it can only overload its own methods.

class Book{

static void print(){}

}

class CourseBook extends Book{

static void print(){}

}//method hiding

-------------------------------

class Book{

static void print(){}

}

class CourseBook extends Book{

void print(){}

}// **compile time error** (instance method cannot override the static method)

-------------------------------------

class Book{

final void print(){}

}

class CourseBook extends Book{

void print(){}

}// **compile time error** (we can’t override final methods)

----------------------------------

class Book{

void print(){}

}

class CourseBook extends Book{

static void print(){}

}//**compile time error** (static method cannot hide the instance method from Book)

------------------------------

class Book{

void print(){}

}

class CourseBook extends Book{

void print(){}

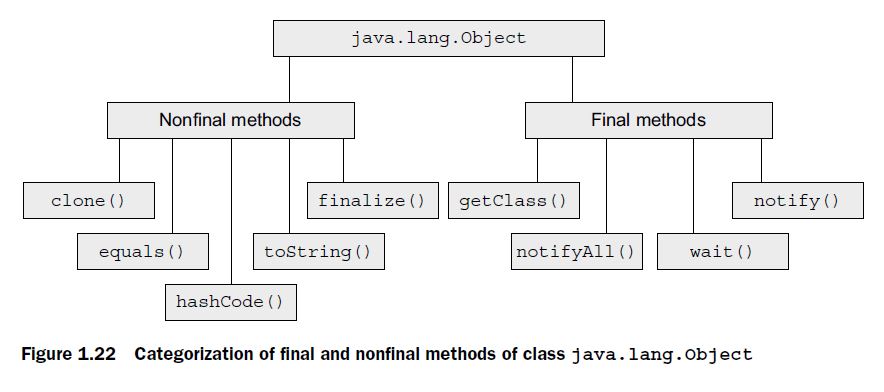
}// perfect overriding

*CAN YOU OVERRIDE BASE CLASS CONSTRUCTORS?*

The simple answer is no. Constructors aren’t inherited by a derived class. Because only inherited methods can be overridden, constructors cannot be overridden by a derived class. If you attempt an exam question that queries you on overriding a base class constructor, you know that it’s trying to trick you.

*EXAM TIP*

Constructors can’t be overridden because a base class constructor isn’t inherited by a derived class.



Following is the code of method toString(), as defined in class Object in the Java API:

public String toString() {

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

*TWIST IN THE TALE 1.1*

Here are the classes written by Shreya and Harry (residing in separate source code files) that work without any issues:

package library; // Class written by Shreya

public class Book {

protected String author;

}

package building; // Class written by Harry

import library.Book;

class StoryBook extends Book {

{ author = "Selvan"; }

}

On Friday evening, Shreya modified her code and checked it in to the organization’s version control system. Do you think Harry would be able to run his code without any errors when he checks out the modified code on Monday morning, and why? Here’s the modified code:

package library; // Class written by Shreya

class Book {

protected String author;

}

package building; // Class written by Harry

import library.Book;

class StoryBook extends Book {

{ author = "Selvan"; }

}

Nonpublic classes can’t be accessible in different package.

RULES FOR OVERRIDING METHOD EQUALS ()

1. “if x.equals(y) returns true, y.equals(x) must return true”.
2. “x.equals(null) should return false”. It shouldn’t throw NPE.

CORRECT AND INCORRECT OVERRIDING OF METHOD EQUALS ()

Note that the type of parameter passed to equals() is Object. Watch out for exam questions that seem to override equals(), passing it to a parameter type of the class in which it’s defined. In the following example, class Course doesn’t override method equals(), it overloads it:

class Course {

String title;

Course(String title) {

this.title = title;

}

public boolean equals(**Course o**) {//**Course doesn’t override toString(), it overloads it.**

return title.equals(o.title);

}

public static void main(String args[]) {

Object c1 = new Course("eJava");

Object c2 = new Course("eJava");

System.out.println(c1.equals(c2));// **Prints “false”**

}

}

EXAM TIP Use Object as the parameter type to equals(). Using any other type will overload equals().

THE NEED TO OVERRIDE METHOD HASHCODE()

Method hashCode() returns a hash-code value for an object, which is used to efficiently store and retrieve values in collection classes that use hashing algorithms, such as HashMap. Hashing algorithms identify the buckets in which they would store the objects and from which they would retrieve them. A well-written method hashCode() ensures that objects are evenly distributed in these buckets. Objects with the same hash-code values are stored in the same bucket. To retrieve an object, its bucket is identified using its hash-code value. If the bucket contains multiple objects, method equals() is used to find the target object.

OVERRIDING METHOD HASHCODE () CORRECTLY

To override method hashCode() correctly, you must follow the below rules:

1. If two objects are equal according to equals(Object)method, then calling hashCode()method on each of the two objects must produce the same integer result.
2. It’s not required that if two objects are unequal according to method equals(Object), that calling method hashCode() on each of the two objects must produce distinct integer results.

EXAM TIP Read the questions on method hashCode() carefully. You might be questioned on incorrect, inappropriate, or inefficient overriding of hashCode(). For Example

class MyNumber {

long number;

MyNumber(long number) {this.number = number;}

public int hashCode() {

return 1654;

}

}

In the preceding code, method hashCode() returns the same hash-code value for all the objects of MyNumber. This essentially stores all the values in the same bucket.

EFFECTS OF USING MUTABLE OBJECTS AS KEYS

Java recommends using immutable objects as keys for collection classes that use the hashing algorithm. What if you don’t?

class MyNumber {

int number;

MyNumber(int number) {this.number = number;}

public int hashCode() {

return number;

}

public boolean equals(Object o) {

if (o != null && o instanceof MyNumber)

return (number == ((MyNumber)o).number);

else

return false;

}

public static void main(String args[]) {

Map<MyNumber, String> map = new HashMap<>();

MyNumber num1 = new MyNumber(2500);

map.put(num1, "Shreya");

num1.number = 100;// *Modify field number of key num1*

System.out.println(map.get(num1));// *Prints “null”—can’t locate object with modified key.*

}

}

Implicit down casting isn’t allowed. You can’t assign reference variables of a base class to reference variables of its derived classes.

EXAM TIP In the absence of explicit casting, you’ll never get ClassCast-Exception—a RuntimeException.

EXAM TIP You can explicitly cast null to any type. It won’t generate a compilation error or throw a ClassCastException. Book book = (Book)null;

EXAM TIP If you cast an instance to a class outside its inheritance tree, you’ll get a compiler error. If you cast an instance to a class within its inheritance tree, but the types don’t match at runtime, the code will throw a ClassCastException.

*USING THE INSTANCEOF OPERATOR*

The *instanceof* operator is used to logically test whether an object is a valid type of a class or an interface. You should proceed with explicit casting only if this operator returns true, or you risk running into a *ClassCastException* at runtime.

EXAM TIP The operator *instanceof* returns false if the reference variable being compared to is null.

EXAM TIP The *instanceof* operator *never* throws a runtime exception; it returns either true or false. If the instanceof operator uses inconvertible types, the code won’t compile.

Course c = new Course();

Student s = new Student();

System.out.println(c instanceof Student);

**//Won’t compile—can’t use instanceof to compare inconvertible types.**

The instanceof operator is preceded by a value (literal value or a variable name) and

is followed by a class, interface, or enum name.

System.out.println(null instanceof Course);

// **Prints “false”—null can’t be an instance of any class.**

EXAM TIP The literal value null isn’t an instance of any class. So <referenceVariable> instanceof <ClassName> will return false.

NOTE The *import* statement doesn’t embed the contents of the imported class in your class, which means that importing more classes doesn’t increase the size of your own class. It lets you use the simple name for a class or interface defined in a separate package.

It’s important to note that you can’t use the import statement to use multiple classes or interfaces with the same names from different packages. For example, the Java API defines class Date in two commonly used packages: java.util and java.sql. To define variables of these classes in a class, use their fully qualified names with the variable declaration:

class AnnualExam {

java.util.Date date1;

java.sql.Date date2;

}

An attempt to use an import statement to import both these classes in the same class will not compile:

import java.util.Date;

import java.sql.Date;

class AnnualExam { }//**Code to import classes with same name from different packages won’t compile**

EXAM TIP Members of a named package can’t access classes and interfaces defined in the *default* package.

STATIC IMPORTS

You can import an individual static member of a class or an interface, or all its static members, by using the import static statement. Though accessible using an instance, the static members are usually accessed by prefixing their name with the class or interface names. By using static import, you can drop the prefix and just use the name of the static variable or method.

To use all **public** (only public variable or methods can accessed in different package) and static members of class ExamQuestion in class AnnualExam without importing each of them individually.

import static certification.ExamQuestion.\*;

REVIEW NOTES

* A top-level class, interface, or enum can only be defined using the public or default access. They can’t be defined using protected or private access.
* Overloaded methods are bound at compile time. Unlike overridden methods they’re bound at runtime.
* Overloaded methods might define a different return type or access or non-access modifier, but they can’t be defined with only a change in their return types or access or non-access modifiers.
* If present, the call to another constructor must be the first statement in a constructor.
* Whenever you intend to override methods in a derived class, use the annotation @Override. It will warn you if a method can’t be overridden properly or if you’re actually overloading a method rather than overriding it.
* Overridden methods can define the same or covariant return types.
* Static methods can’t be overridden. They’re not polymorphic and they’re bound at compile time.
* In a derived class, a static method with the same signature as that of a static method in its base class hides the base class method.
* A derived class can’t override the base class methods that aren’t accessible to it, such as private methods. Similarly, base constructor is not part of inheritance or base constructor can’t be accessible in derived class, so we can’t override constructor’s but we can overload constructor’s.
* The members of the default package are accessible only to classes or interfaces defined in the same directory on your system. A class from the default package can’t be used in any named packages.
* An import statement can’t be placed before a package statement in a class. Any attempt to do so will cause the compilation of the class to fail.
* The declaration of private variables to store the state of an object is encouraged.
* Expose the functionality of your classes using public methods.
* Create private methods to work as helper methods for the public methods.
* An instance of a non-final class can be casted to **any interface type** using an explicit cast during the compilation phase. But the exact object types are validated during runtime and a ClassCastException is thrown if the object’s class doesn’t implement that interface.

class Bird{}

class Parrot extedns Bird{}

Parrot p = (Parrot)bird;// ClassCastException

class Bird{}

class Parrot{}

Bird b = (Bird)parrot;//Compilation fail’s because objects are in different hierarchy

interface Scavenger{}

class Bird{}

Scavenger sc2 = (Scavenger)bird;// no compilation error, but ClassCastException

EXAM TIP An abstract method doesn’t define an implementation. It enforces all the concrete derived classes to implement it.

*Advanced class design*

* Should my application be allowed to create instances of Animal class? If no, define Animal class as an abstract class.
* Does class Animal include behavior that’s common to all its derived classes? If yes, define the relevant method as an abstract method and class Animal as an abstract class.

EXAM TIP An abstract method doesn’t define an implementation. It enforces all the concrete derived classes to implement it.

EXAM TIP Abstract classes make a point loud and clear: they *force* the concrete derived classes to implement a base class’s abstract methods, in their own unique manner.

DON’T CREATE AN ABSTRACT CLASS ONLY TO PREVENT CREATION OF ITS OBJECTS

To prevent instantiation of a class by using the operator new, define all its class constructors as private.

DON’T CREATE OBJECTS OF AN ABSTRACT CLASS

Code that creates objects of an abstract class won’t compile:

abstract class Animal{}

class Forest {

Animal animal = new Animal();**Won’t compile;**

}

DON’T DEFINE AN ABSTRACT CLASS AS A FINAL CLASS

A final class can’t be extended. On the other hand, abstract classes are created so they can be extended by other classes. Hence, abstract classes can’t be defined as final classes.

abstract final class Animal {}// **Won’t compile**

*STATIC MODIFIER*

You can define variables, methods, nested classes, and nested interfaces as static members. They belong to a class and not to instances. They can be accessed soon after their class is loaded into memory. Top-level classes, interfaces, and enums can’t be defined as static entities. Watch out for code that declares top-level classes, interfaces, and enums as static members. Such code won’t compile.

EXAM TIP Unlike instance variables, which are initialized for each instance, static class variables are initialized only once, when they are loaded into JVM memory. The default variable values are false for boolean; '\u0000' for char; 0 for byte; short, int, 0L for long; 0.0F for float; 0.0D for double; and null for objects.

EXAM TIP You can access a static member by using the name of its class or any of its instances. All these approaches refer to the same static member. The preferred approach is to use a class name; otherwise, a static member *seems* to be tied to an instance, which is incorrect.

The static methods can’t access non-static fields or non-static methods directly.

**static int x = count();**

int count() { return 10; }

// Compilation error. Static variable or methods should not access instance variables or methods directly.

A static initializer block is used to initialize static variables. This initializer block executes when a class is loaded by the JVM into memory. You can define multiple static initializer blocks in your code, which execute in the order of their appearance. Static variables can’t be initialized using the constructors of a class.

class AffiliateProgram {

private static int accountOpenBonus;

static {

accountOpenBonus = 5;

}

}

EXAM TIP On the exam, beware of code that defines multiple initializer blocks. If a class defines multiple initializer blocks, they execute in the order of their appearance in a class.

* You can’t prefix the definition of a top-level class or an interface with the keyword static. But define inner class or interface as static.

EXAM TIP If a static or instance variable is marked final, it must be initialized, or the code won’t compile.

* Interestingly, you can survive code with an uninitialized final local variable, if you don’t use it.
* To initialize final static variable we can use static block, to initialize final variable we can use constructor.

Method parameters are initialized when the method is invoked. If a method marks its method parameter(s) as final, the method body can’t reassign a value to it, as follows:

class MyClass {

void setValue(**final** int finalMethodParam) {

finalMethodParam = 10;

}

}

There’s a difference between final primitive variables and final object reference variables. The final primitive variables can’t change, but the object referred to by final object reference variables can be changed. Only the final reference itself can’t be changed:

class MyClass {

void addCondition(final StringBuilder query) {

query.append("WHERE id > 500");

query = new StringBuilder("SELECT name FROM emp");// **Won’t compile**

}

}

EXAM TIP On the exam, look out for multiple initializations of a final variable. Code snippets that try to reinitialize a final variable won’t compile.

As you know, you can override only what is inherited by a derived class. The following code compiles successfully, even though the derived class *seems* to override a final method from its base class:

class Base {

private final void finalMethod() {}// **Private methods aren’t inherited.**

}

class Derived extends Base {

final void finalMethod() {}

}

EXAM TIP The private methods of a base class aren’t inherited by its derived classes. A method using the same signature in the derived class isn’t an overriding method, but a new method.

EXAM TIP Look out for trick questions on the exam that extend final classes from the Java API, like class String and the wrapper classes Byte, Short, Integer, Long, Float, Double, Boolean, and Character. When you don’t look at the source code of the base class and see that it’s marked final, it’s easy to overlook that the classes that extend it won’t compile. Though the authors of this exam claim not to include trick questions, they also state that they expect the candidates to know “their stuff.”

***Enumerated types***

Users are allowed to use only *existing* enum objects; they can’t create new enum objects. ***Type safety***was the main reasonfor introducing enumerated types in Java version 5.0.

***Understanding the need for and creating an enum***

Let’s assume that you have been assigned the task of creating a gaming application that can be played at exactly three levels: beginner, intermediate, and expert. How would you restrict your variable to be assigned only these three values? You can accomplish this by creating an enum. An enum enables you to create a *type*, which has a *fixed* set of *constants*. Following is an example of the enum Level, which defines three programming levels:

enum Level { BEGINNER, INTERMEDIATE, EXPERT }

-----------------------------------------------------------------------

final class Level extends Enum// **enum is implicitly declared as final.**

{

public static final Level BEGINNER;

public static final Level INTERMEDIATE;

public static final Level EXPERT;

// **enum constants are implicitly public, static, and final.**

private static final Level $VALUES[];

//**Array to store reference to all enum constants**

static

{

BEGINNER = new Level("BEGINNER", 0);

INTERMEDIATE = new Level("INTERMEDIATE", 1);

EXPERT = new Level("EXPERT", 2);

$VALUES = (new Level[] {

BEGINNER, INTERMEDIATE, EXPERT

});

}

// **Creation of enum constants occurs in static initializer block**

public static Level[] values()

{

return (Level[])$VALUES.clone();

}

// **return an array of all enum constants.**

public static Level valueOf(String s)

{

return (Level)Enum.valueOf(Level, s);

}

// **Method valueOf() parses a String value and returns corresponding enum constant**

private Level(String s, int i)

{

super(s, i);

}

// **Private constructor**

}

All enums in Java extend the abstract class java.lang.Enum, defined in the Java API.

public abstract class Enum<E extends Enum<E>>

implements Comparable<E>, Serializable {

private final String name;

private final int ordinal;

protected Enum(String name, int ordinal) {

this.name = name;

this.ordinal = ordinal;

}

public String toString() {

return name;

}

// **Default implementation of toString() returns name of enum constant**

public final String name() {

return name;

}

//.. rest of the code

}

// **Method name() is marked final and can’t be overridden; returns enum constant’s name.**

Note both methods—toString() and name() defined in java.lang.Enum—return the value of the instance variable name. Because method name() is a final method, you can’t override it. But you can override method toString() to return any description that you want.

EXAM TIP Watch out for exam questions that use methods like Collections.sort() from the Collections API to sort enum constants. The default order of enum constants is their order of definition. The enum constants aren’t sorted alphabetically.

Because a class can extend from only one base class, an attempt to make your enum extend any other class will fail. The following code won’t compile:

class Person {}

enum Level **extends Person** { BEGINNER, INTERMEDIATE, EXPERT }

But you can make your enum implement any number of interfaces.

You can’t explicitly make a class extend java.lang.Enum:

class MyClass extends java.lang.Enum {}

EXAM TIP An enum implicitly extends java.lang.Enum, so it can’t extend any other class. But a class can’t explicitly extend java.lang.Enum.

EXAM TIP The enum constant list must be the first in the enum definition and should be followed by a semicolon. A semicolon is optional if you don’t add methods and variables to your enum.

EXAM TIP An enum can’t define a constructor with public or protected access level. We can define default or private.

EXAM TIP An enum constant can define a constant specific class body and use it to override existing methods or define new variables and methods.

Rules to remember about enums

* An enum can define a main method. This means that you can define an enum as an executable Java application.
* The enum constant list must be defined as the first item in an enum, before the declaration or definition of methods and variables.
* The enum constant list might not be followed by a semicolon, if the enum doesn’t define any methods or variables.
* When an enum constant overrides an enum method, the enum constant creates an anonymous class, which extends the enum.
* An enum implicitly extends java.lang.Enum, so it can’t extend any other class. But a class can’t explicitly extend java.lang.Enum. An enum can implement interface(s).
* An enum can never be instantiated using the keyword new.
* You can define multiple constructors in your enums.
* An enum can define an abstract method. Just ensure to override it for all your enum constants.
* The enum method values() returns a list of all the enum constants.
* An enum can be defined as a top-level enum, or as a member or another class or interface. It can’t be defined local to a method.

***Static nested and inner classes***

To access the static members of a static nested class, you need not create an object of this class. You need an object of a static nested class to access its instance members.

Here’s an example:

class Outer1 {

public static void main(String args[]) {

System.out.println(new Outer.StaticNested().innerInstance);

//**Object of StaticNested class required to access its instance members**

System.out.println(Outer.StaticNested.innerStatic);

//**Object of StaticNested class not required to access its static members**

}

}

A static nested class can be defined using all access levels: private, *default* access, protected, and public. The accessibility of the static nested class depends on its access modifier. For example, a private static nested class can’t be accessed outside its outer class. The access of a static nested class also depends on the accessibility of its outer class. If the outer class is defined with the default access, an inner nested class with public access won’t make it accessible outside the package in which its outer class is defined.

A static nested class can access only the static members of its outer class. Can’t access instance variables from a static nested class directly.

Rules to remember about static nested classes

* To create an object of a static nested class, you need to prefix its name with the name of its outer class.
* A static nested class can define both static and non-static members. But non static inner class can define only non-static members.
* You need not create an object of a static nested class to access its static members.
* They can be accessed the way static members of a regular class are accessed.
* You should create an object of a static nested class to access its non-static members, by using the operator new.
* A static nested class can be defined using any access modifier.
* A static nested class can define constructor(s).

EXAM TIP You *must* have an outer class instance to create an inner class instance.

EXAM TIP The accessibility of an *inner class* outside its *outer class* depends on the access modifier used to define the inner class. For example, an inner class with default access can’t be accessed by classes in different packages than the outer class.

Rules to remember about inner classes

* You can create an object of the inner class within an outer class or outside an outer class.
* When an inner class is created outside its outer class, its type name should include the name of its outer class, followed by a dot (.) and then the name of the inner class.
* To create an inner class with in **static method of an outer class**, or outside an outer class, call the operator new on the object of the outer class to instantiate the inner class.
* An inner class can’t define **static methods**. It can define **final static variables** but **non-final static variables** aren’t allowed.
* Members of the inner class can refer to all variables and methods of the outer class.
* An inner class can be defined with all access modifiers.
* An inner class can define constructors.
* An inner class can define variables and methods with any access level.

Inner classes save you from defining new classes.

EXAM TIP An anonymous inner class can extend at most one class or implement one interface. Unlike other classes, an anonymous class can neither implement multiple interfaces, nor extend a class and implement an interface together.

By using an anonymous class, you can override the methods from its base class or implement the methods of an interface. You can also define new methods and variables in an anonymous class.

interface Flyable{

void fly();

}

class BirdSanctuary {

Flyable bird = new Flyable(){

public void fly() {

System.out.println("Flying high in the sky");

}

**public void hungry(){**

**System.out.println("eat");**

**}**

};

}

You can’t call the additional member, method hungry(), using the reference variable bird. Why? The type of the reference variable bird is Flyable. So the variable bird can access only the members defined in interface Flyable. The variable bird can’t access additional methods and variables that are defined in anonymous classes that implement it.

For Inner and static Inner classes: Outer$Inner.class, and Outer$Inner.class.

For method/local Inner class: Outer$1Inner.class, and Outer$2Inner.class.

A method local inner class can access all variables and methods of its *outer class*, including its private members and the ones that it inherits from its base classes. But a method local inner class can’t define static variables or static methods. Method local inner can’t be defined using an explicit access modifier always default.

* You can define final static variables in a method local inner class, but you can’t define non-final static variables, static methods, or static final methods. You can define constructors with any access modifier in a local inner class.
* You can’t define an enum within a method or a non-static inner class.
* An anonymous inner class always extends a class, implicitly (When it implements an interface, it implicitly extends class java.lang.Object.) or explicitly.
* You can’t make an anonymous class implement multiple interfaces explicitly.

***Abstract classes***

* An *abstract class* is defined by using the keyword abstract. It defines variables to store the state of an object. It may define abstract and non-abstract methods.
* An abstract class must not necessarily define abstract methods. But if it defines even one abstract method, it must be marked as an abstract class.
* An abstract class can’t be instantiated.
* An abstract method doesn’t have any implementation. It represents a behavior that’s required by all derived classes of an abstract class. Because the base class doesn’t have enough details to implement an abstract method, the derived classes are left to implement it in their own specific manner.
* An abstract class can’t be instantiated.
* An abstract class *forces* all its non-abstract-derived classes to implement the incomplete functionality in their own unique manner.
* A base class should be defined as an abstract class so it can implement the available details but still prevent itself from being instantiated.
* An abstract class can be extended by both abstract and concrete classes. If an abstract class is extended by another abstract class, the derived abstract class *might* not implement the abstract methods of its base class.
* If an abstract class is extended by a concrete class, the derived class *must* implement all the abstract methods of its base class, or it won’t compile.
* A derived class must call its superclass’s constructor (implicitly or explicitly), irrespective of whether the superclass or derived class is an abstract class or concrete class.
* An abstract class can’t define abstract static methods. Because static methods belong to a class and not to an object, they aren’t inherited. A method that can’t be inherited can’t be implemented. Hence this combination is invalid.
* Efficient use of an abstract class lies in the identification of an abstract class in your application design so you can define common code for your objects and leave the ones that are more specific, by defining them as abstract. You can enforce the derived classes to implement these abstract methods.

***Non-access modifier—static***

* Static members (fields and methods) are common to all instances of a class, and aren’t unique to any instance of a class.
* Static members exist independently of any instances of a class, and may be accessed even when no instances of the class have been created.
* Static members are also known as *class fields* or *class methods* because they are said to belong to their class, and not to any instance of that class.
* A static variable and method can be accessed using the name of an object reference variable or the name of a class.
* A static method and variable can’t access non-static variables and methods of a class. But the reverse works: non-static variables and methods can access static variables and methods.
* Static classes and interfaces are a type of nested classes and interfaces.
* You can’t prefix the definition of a top-level class or an interface with the keyword static. A top-level class or interface is one that isn’t defined within another class or interface.

***Non-access modifier—final***

* You can’t reinitialize a final variable defined in any scope—class, instance, local, or method parameter.
* An instance final variable can be initialized either with its declaration in the initializer block or in the class’s constructor.
* A static final variable can be initialized either with its declaration or in the class’s static initializer block.
* You can’t initialize a final instance variable in an instance method because it can’t be guaranteed to execute only once. Such a method won’t compile.
* You can’t initialize a final static variable in a static method because it can’t be guaranteed to execute only once. Such a method won’t compile.
* If you don’t initialize a final local variable in a method, the compiler won’t complain, as long as you don’t use it.
* If you try to access the value of a final local variable before assigning a value to it, the code won’t compile.
* The Java compiler considers initialization of a final variable complete *only* if the initialization code will execute in *all* conditions. If the Java compiler can’t be sure of execution of code that assigns a value to your final variable, it will complain (code won’t compile) that you haven’t initialized a final variable. If an if construct uses constant values, the Java compiler can predetermine whether the then or else blocks will execute. In this case, it can predetermine whether these blocks of code will execute to initialize a final variable.
* A final instance variable defined in a base class can’t be initialized in the derived class. If you try to do so, your code won’t compile.
* Final methods defined in a base class can’t be overridden by its derived classes.
* Final methods are used to prevent a derived class from overriding the implementation of a base class’s method.
* Private final methods in a base class aren’t inherited by derived classes. A method defined using the same method signature in a derived class isn’t an overridden method, but a new method.
* A final class can’t be extended by any other class.
* A class is defined as final so that it can’t be extended by any other class. This prevents objects of derived classes from being passed on to reference variables of their base classes.
* An interface can’t be defined as final because an interface is abstract, by default. A Java entity can’t be defined both as final and abstract.

***Enumerated types***

* Enumerated types are also called *enums*.
* An enum enables you to create a *type*, which has a *fixed* set of *constants*.
* An enum can never be instantiated using the keyword new.
* Unlike a class, which is defined using the keyword class, an enumerated type is defined using the keyword enum, and can define multiple variables and methods.
* If you define a variable of an enum type, it can be assigned constant values only from that enum.
* All enums extend the abstract class java.lang.Enum, defined in the Java API.
* Because a class can extend from only one base class, an attempt to make your enum extend any other class will fail its compilation.
* The enum constants are implicit static members.
* An enum can implement any interface, but its constants should implement the relevant interface methods.
* An enum can define an abstract method. Just ensure that you override it for all your enum constants.
* You can add instance variables, class variables, instance methods, and class methods to your enums.
* An enum can’t use instance variables in the overridden methods for a particular enum constant.
* You can override nonfinal methods from class java.lang.Enum, for individual (or all) enum constants.
* Your enums can also define constructors, which can be called from within the enum.
* You can define multiple constructors in your enums.
* Enum constants can define new methods, but these methods can’t be called on the enum constant.
* You can define an enum as a top-level enum or within another class or interface.
* You can’t define an enum local to a method.
* An enum can define a main method.

***Static nested classes***

* This class isn’t associated with any object of its outer class. Nested within its outer class, it’s accessed like any other static member of a class—by using the class name of the outer class.
* A static nested class is accessible outside the class in which it’s defined by using names of both the outer class and inner class.
* You can define both static and non-static members in a static nested class.
* A static nested class can define constructors.
* To access the static members of a static nested class, you need not create an object of this class. You need an object to access the instance members of this class.
* The accessibility of the nested static class depends on its access modifier. For example, a private static nested class can’t be accessed outside its class.
* A static nested class can access only the static members of its outer class. Similarly, the outer class can access only the static members of its nested inner class.
* An attempt to access instance members on either side will fail compilation unless it’s accessed through an instance of the outer or static nested class.
* All access levels can be used with this class—public, protected, *default*, and private.

***Inner classes***

* An inner class is an *instance member* of its outer class.
* An object of an *inner class* shares a special bond with its *outer class* and can’t exist without its instance.
* An inner class can be defined using any of the four access levels—public, protected, *default*, and private.
* Members of an inner class can refer to all variables and methods of an outer class.
* An inner class can define constructors.
* An inner class can define variables and methods with any access.
* An inner class can’t define static methods and non-final static variables.
* You can create an object of an inner class within an outer class or outside an outer class.
* Outside the outer class an inner class is instantiated using

**Outer.Inner** inner = **new** Outer().**new** Inner();

***Anonymous inner classes***

* An anonymous inner class is created when you combine object instance creation with inheriting a class or implementing an interface.
* An anonymous inner class might override none, few, or all methods of the inherited class.
* An anonymous inner class must implement all methods of the implemented interface.
* An instance of an anonymous class can be assigned to any type of variable (static variable, instance variable, or local variable) or method parameter, or be returned from a method.
* The following line creates an anonymous inner class that extends Object and assigns it to a reference variable of type Object:

Object obj = new Object(){};

* The following line calls a method, say aMethod(), passing to it an instance of an

anonymous class that implements Runnable:

aMethod(new Runnable() {

public void run() {}

});

* When an anonymous inner class is defined within a method, it can access only the final variables of the method in which it’s defined. This is to prevent reassignment of the variable values by the inner class.
* Though you can define variables and methods in an anonymous inner class, they can’t be accessed using the reference variable of the base class or interface, which is used to refer to the anonymous class instance.

***Method local inner classes***

* Method local inner classes are defined within a static or instance method of a class.
* A class can define multiple method local inner classes, with the same class name, but in separate methods.
* Method local inner classes can’t be defined using any explicit access modifier.
* A method local inner class can define its own constructors, variables, and methods by using any of the four access levels—public, protected, *default*, and private.
* A method local inner class can be created only within the method in which it’s defined. Also, its object creation can’t appear before its declaration.
* A method local inner class can access all variables and methods of its *outer class*, including its private members and the ones that it inherits from its base classes. It can only access the final local variables of the method in which it’s defined.
* A method local inner class can define members with the same name as its outer class. In this case, the members of the outer class can be referred to by using Outer.this.

***Object-oriented design principles***

interface Runner{

int speed();

double distance = 70;

}

Becomes

interface Runner{

public abstract int speed();

public static final double distance = 70;

}

All methods of an interface are implicitly public and abstract, and its variables are implicitly public, static, and final. Why do you think these implicit modifiers are added to the interface members? Because an interface is used to define a contract, it doesn’t make sense to limit access to its members—and so they are implicitly public. An interface can’t be instantiated, and so the value of its variables should be defined and accessible in a static context, which makes them implicitly static. Because an interface is a contract, its implementations shouldn’t be able to change it, so the interface variables are implicitly final. Interface methods are implicitly abstract so that it’s mandatory for the classes to implement them.

EXAM TIP The declaration of an interface can’t include a class name. An interface can never extend any class.

Can you define a top-level, *protected* interface? No, you can’t.

You can declare a *top-level interface,* with only the following access levels:

* Public
* No modifier (default access)

EXAM TIP All the top-level Java types (classes, enums, and interfaces) can be declared using only two access levels: public and default. Inner or nested types can be declared using any access level.

All members of an interface—variables, methods, inner interfaces, and inner classes are public by default. Interfaces support only the public access modifier. Using other access modifiers results in compilation errors.

Because the interface variables are implicitly final, you can define only *constants* in an interface. Ensure that you initialize these constants, or your code won’t compile:

interface MyInterface {

int number; **//Won’t compile;**

}

You can declare a top-level interface with only the following nonaccess modifiers:

* Abstract
* strictfp

NOTE The strictfp keyword guarantees that results of all floating-point calculations are identical on all platforms.

If you try to declare your top-level interfaces by using the other nonaccess modifiers (final, static, transient, synchronized, or volatile), the interface will fail to compile. All of the following interface declarations fail to compile:

EXAM TIP A *concrete* class must implement all the methods from the interfaces that it implements. An *abstract* class can choose not to implement all the methods from the interfaces that it implements.

A class can define an instance or a static variable with the same name as the variable defined in the interface that it implements.

interface Livable {

boolean status = true;

int ratings = 10;

}

class Home implements Livable {

boolean status;

static int ratings = 7;

Home() {

System.out.println(status);//**false**

System.out.println(Livable.status);//**true**

System.out.println(ratings);//7

System.out.println(Livable.ratings);//11

}

}

EXAM TIP A class can define an instance or a static variable with the same name as the variable defined in the interface that it implements. These variables can be defined using any access level.

EXAM TIP Because interface methods are implicitly public, the implementing class must implement them as public methods, or else the class will fail to compile.

EXAM TIP A class can implement methods with the same name from multiple interfaces. But these must qualify as correctly overloaded methods.

* A class can *implement* multiple interfaces. An interface can *extend* multiple interfaces.
* If a class implements multiple interfaces that define methods with the same name, the interface methods must either qualify as correctly overloaded or overridden methods, or else the class won’t compile.

FRAGILE DERIVED CLASSES

Adding to or modifying a base class can affect its derived classes. Adding new methods to a base class can result in breaking the code of a derived class. Consider this initial arrangement, which works well:

public abstract class Animal {

void move(){}

}

class Lion extends Animal {

void live(){}

}

Now consider a modified arrangement: a new method live() is added to base class Animal. Because live() clashes (because of an incorrectly overridden method) with the existing method live() in its derived class Lion, Lion will no longer compile:

public abstract class Animal {

void move(){}

String live(){

return "live";

}

}

class Lion extends Animal {

void live(){}//**live() in Lion neither overloads nor overrides live() in Animal.**

}

*Preferring Class Inheritance Over Interface Inheritance*

Class inheritance scores better when you want to reuse the implementation already defined in a base class. It also scores better when you want to add new behavior to an existing base class.

*Preferring interface inheritance over class inheritance*

You may prefer interface inheritance over class inheritance when you need to define multiple contracts for classes.

* IMPLEMENTING MULTIPLE INTERFACES
* FRAGILE DERIVED CLASSES

EXAM TIP Class inheritance isn’t always a good choice because derived classes are fragile. If any changes are made to a base class, a derived class might break. Extending classes that are from another package or are poorly documented aren’t good candidates for base classes.

If a base class chooses to modify the implementation details of its methods, the derived classes might not be able to offer the functionality they were supposed to, or they might respond differently. Consider this initial arrangement:

public abstract class Animal {

String currentPosition;

public void move(String newPosition){

currentPosition = newPosition;

}

}

class Lion extends Animal {

void changePosition(String newPosition) {

super.move(newPosition);

System.out.println("New Position:" + newPosition);

}

}

class Test{

public static void main(String args[]) {

new Lion().changePosition("Forest");

}

}

Imagine that Animal adds another line of code to method move(). Let’s see how it changes the code output of class Test (modification in bold):

public abstract class Animal {

String currentPosition;

public void move(String newPosition){

currentPosition = newPosition;

**System.out.println("New Position:" + newPosition);**

}

}

class Lion extends Animal {

void changePosition(String newPosition) {

super.move(newPosition);

System.out.println("New Position:" + newPosition);

}

public static void main(String args[]) {

new Lion().changePosition("Forest");

**//Prints “New Position:Forest” twice**

}

}

As you witnessed in the preceding example, changes to a base class can break the code of its derived classes.

EXAM TIP There isn’t any clear winner when it comes to selecting the better option from class inheritance and interface inheritance. Analyze the given conditions or situations carefully to answer questions on this topic.

EXAM TIP Representing IS-A and HAS-A relationships by using (quick) UML diagrams can help you on the exam. Though you may not see UML diagrams on the exam, creating quick UML diagrams on an erasable board (or something similar) provided to you during the exam will help you answer these questions.

EXAM TIP The key to finding the types that participate in an IS-A relationship is to find your way, up the hierarchy tree, in the direction of the arrows. This technique will not only help you with the exam, but also take you a long way in your professional career.

EXAM TIP Well-designed applications aim for loosely coupled classes and modules.

EXAM TIP Object composition enables you to use the existing functionality of classes without extending them. The approach is simple: create and use objects of other classes in your own class.

You should inherit a class only when you think that the derived class is a type of its base class. For example, it’s correct to say that RacingCar is a type of Car. But it’s incorrect to say that Engine is a type of Car.

EXAM TIP On the exam, all of these approaches (eager initialization, synchronization of the complete method getInstance(), and partial synchronization of method getInstance()) may be presented, and you may be questioned about the right approach for implementing the Singleton pattern. All these approaches are good. Beware of modified code that tries to synchronize a partial getInstance() method, which doesn’t synchronize the code that creates an object of Singleton.

USING ENUMS

Enum in Javais a keyword, a feature which is used to represent fixed number of well-known values in Java, For example, Number of days in Week, Number of planets in Solar system etc.

One of the common use of Enum which emerged in recent years is [Using Enum to write Singleton in Java](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html), which is by far easiest way to implement Singleton and handles several issues related to thread-safety and Serialization automatically.

By using enums, you can implement the Singleton pattern in a thread-safe manner.

Here’s a simple implementation:

public enum Singleton {

INSTANCE;

public void initCache(){

//..code

}

}

Because enum instances can’t be created by any other class, the enum Singleton will ensure the existence of only *one* of its instances, Singleton.INSTANCE.

NOTE The Singleton pattern is also referred to as an anti-pattern. It has been overused by developers and designers, who make a lot of assumptions about the applications that use it. **It also makes testing difficult**.

* Prefer method invocation over direct constructor calls
* Promote flexibility. Object instantiation logic can be changed without affecting the clients that use objects. They also allow addition of new concrete classes.

|  |  |  |
| --- | --- | --- |
| CLASS | METHOD | DESCRIPTION |
| java.util.Calendar | getInstance() | Gets a calendar using the default time zone and locale. |
| java.util.Arrays | asList() | Returns a fixed-size list backed by the specified array. |
| java.util.ResourceBundle | getBundle() | Overloaded versions of this method return a resource bundle using the specified base name, target locale, class loader, and control. |
| java.sql.DriverManager | getConnection() | Establishes and returns a connection to the given database URL. |
| java.sql.DriverManager | getDriver() | Attempts to locate and return a driver that understands the given URL. |
| java.sql.Connection | createStatement() | Overloaded version of this method creates a statement object for sending SQL statements to the database and generates ResultSet  objects with the given type, concurrency, and holdability. |
| java.sql.Statement | executeQuery() | Executes the given SQL statement, which returns a single ResultSet object. |
| java.text.NumberFormat | getInstance()  getNumberFormat() | Returns a general purpose number format for the current default locale. |
| java.text.NumberFormat | getCurrencyInstance() | Returns a currency format for the current default locale. |
| java.text.NumberFormat | getIntegerInstance() | Returns an integer format for the current default locale. |
| java.util.concurrent  .Executors | newFixedThreadPool()  newCachedThreadPool()  newSingleThreadExecutor() | Creates a thread pool. |

EXAM TIP The DAO pattern decouples classes that define business or presentation logic from the data persistence details.

EXAM TIP The exam might ask you whether it’s common to use the Factory pattern with the DAO pattern. The answer is yes (as shown in the next section). But it isn’t mandatory to use the Factory pattern with the DAO pattern (as shown in this section).

***Benefits of the DAO pattern***

The benefits of the DAO pattern are

* It abstracts and encapsulates all access to a data source. It manages the connection to the data source to obtain and store data.
* It promotes programming to an interface. It completely hides the data access implementation from its clients.
* It decouples the business logic layer and persistence layer. It makes the code independent of any changes to a data source or its vendor (for example, plaintext, XML, LDAP, MySQL, Oracle, or DB2).
* It promotes flexibility. Because the interfaces accessible to client classes don’t change, new implementation classes can be added.
* The DAO pattern might also include Factory pattern classes.
* It prevents tight coupling between client classes and DAO implementation classes. It promotes the creation of cohesive classes.

***Summary***

* Class inheritance also scores better when you want to add new behavior to an existing base class. You may prefer interface inheritance over class inheritance when you need to define multiple contracts for classes.

***Interfaces***

* An interface is an example of separating the behavior that an object should support from its implementation. An interface is used to define behavior by defining a group of abstract methods.
* All members (variables and methods) of an interface are implicitly public.
* You declare an interface using the keyword interface. An interface can define only public, final, static variables and public, abstract methods.
* The methods of an interface are implicitly abstract and public.
* The variables of an interface are implicitly public, static, and final.
* You can declare a top-level interface only with public and default access. Valid non-access modifiers that can be applied to an interface are abstract and strictfp.
* An interface that’s defined within another interface can be defined with any access modifier.
* An interface can’t extend a class.
* An interface can extend multiple interfaces. It can’t implement another interface.
* An interface can define inner interfaces and (surprisingly) inner classes too.
* Because all the members of an interface are implicitly public, a derived interface inherits all the methods of its base interface.
* You can compare interface implementation to the signing of a contract. When a concrete class declares an implementation of an interface, it agrees to and must implement all its abstract methods.
* If you don’t implement all the methods defined in the implemented interfaces, a class can’t compile as a concrete class. A concrete class must implement all the methods from the interfaces that it implements. An abstract class might not implement all the methods from the interfaces that it implements.
* A class can define an instance or a static variable with the same name as the variable defined in the interface that it implements. These variables can be defined using any access level.
* Because the methods in an interface are implicitly public, if you try to assign a weaker access to the implemented method in a class, it won’t compile.
* A class can inherit methods with the same name from multiple interfaces. There are no compilation issues if these methods have exactly the same method signature or if these methods can coexist in the implemented class as overloaded methods. The class won’t compile if these methods coexist as incorrectly overloaded or overridden methods.

***Class inheritance versus interface inheritance***

* Class inheritance scores better when you want to reuse the implementation already defined in a base class. It also scores better when you want to add new behavior to an existing base class.
* You can add new behavior to an abstract or non-abstract base class, and you may not break all the classes that subclass it.
* You may prefer interface inheritance over class inheritance when you need to define multiple contracts for classes.
* Interface implementation has one major advantage of allowing a class to implement multiple interfaces, so an object of the class can be assigned to variables of multiple interface types.

***Object composition principles***

* Newcomers to programming often extend a class when they want to use a class in another class. They use inheritance in place of composition.
* You should extend a class (inheritance) when you want the objects of the derived classes to reuse the interface of their base class.
* You should define an object of another class (composition) when you want to use the functionality offered by the class.

***Singleton pattern***

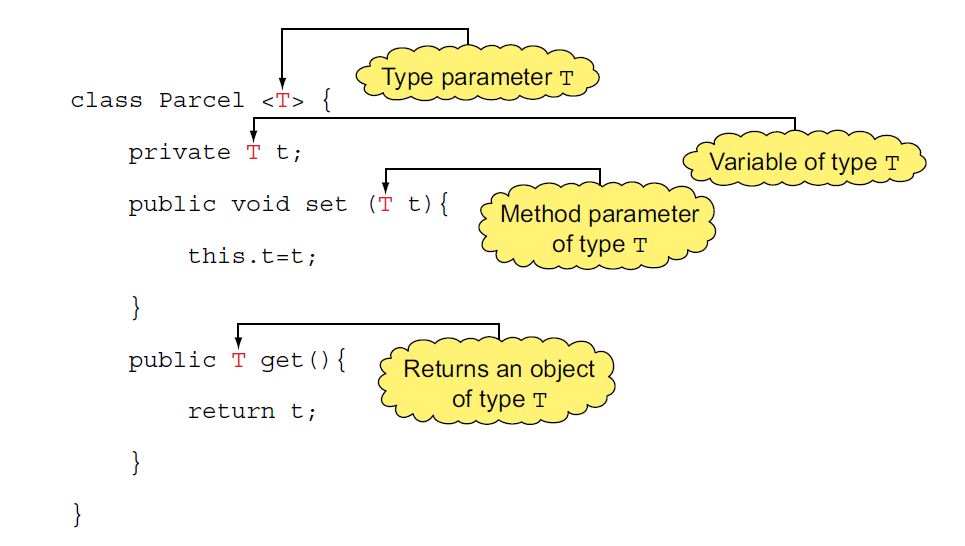
* Singleton is a creational design pattern that ensures that a class is instantiated only once. The class also provides a global point of access to it.
* It is used in scenarios when you might need only one object of a class.
* Implementation of the Singleton pattern involves a single class.
* A class that implements the Singleton pattern must define its constructor as private.
* A Singleton class uses a static private reference variable to refer to its sole instance.
* A Singleton class defines a static method to access its sole instance.
* You can also use enums to implement the Singleton pattern because enum instances can’t be created by any other class.
* To avoid threading issues with the creation of the sole instance of the Singleton class, you might use either of the following to create its sole instance:
* Eager initialization—instantiate the object with its declaration
* Synchronized lazy initialization—create the instance using a synchronized method or code block

***Factory pattern***

* One of the most frequently used design patterns, multiple flavors of this pattern exist: Simple Factory, Factory Method, and Abstract Factory.
* The Simple Factory pattern creates and returns objects of classes that extend a common parent class or implement a common interface. The objects are created without exposing the instantiation logic to the client. The calling class is decoupled from knowing the exact name of the instantiated class.
* The intent of the Factory Method pattern is to define an interface for creating an object but let subclasses decide which class to instantiate. The Factory Method pattern lets a class defer instantiation to its subclasses.
* The Abstract Factory pattern is used to create a family of related products (in contrast, the Factory Method pattern creates one type of object). This pattern also defines an interface for creating objects but it lets subclasses decide which class to instantiate.
* The benefits of the Factory pattern are
* Prefers method invocation over direct constructor calls
* Prevents tight coupling between a class implementation and your application
* Promotes creation of cohesive classes
* Promotes programming to an interface
* Promotes flexibility. Object instantiation logic can be changed without affecting the clients that use objects. It also allows the addition of new concrete classes.

*Generics and collections*

Generic in Java is added to provide compile time type-safety of code and removing risk of ClassCastException at [runtime](http://javarevisited.blogspot.sg/2012/03/what-is-static-and-dynamic-binding-in.html) which was quite frequent error in Java code.  
EXAM TIP The basic purpose behind using generics is to enable you to mark your intent of using a class, method, or interface with a particular data type. Generics add compile-time safety to collections.



EXAM TIP The first occurrence of T is different from its remaining occurrences because only the first one is surrounded by <>.

EXAM TIP A type parameter can be used in the declaration of classes, variables, method parameters, and method return types.

GENERIC CLASS EXTENDING ANOTHER GENERIC CLASS

A generic class can be extended by another generic class. In the following example, generic class GenericBookParcel<T> extends generic class Parcel<T>:

class Parcel<**T**> {}

class GenericBookParcel<**T**> extends Parcel<**T**> {}

In all cases, an extended class must be able to pass type arguments to its base class. For the preceding example, the type argument passed to class GenericBookParcel is passed to its base class, Parcel, when you instantiate GenericBookParcel. For example

GenericBookParcel<String> parcel = new GenericBookParcel<>();

The preceding example passes argument String to GenericBookParcel’s type parameter T. But if you define GenericBookParcel in a way that it can’t pass an argument to the parameters of its base class, the code won’t compile. Do you think the following code will compile?

class Parcel<**T**> {}

class GenericBookParcel<**X**> extends Parcel<**T**> {}

// **Won’t compile; no way to pass argument to T**

No, it won’t. In the preceding code, class GenericBookParcel defines a type parameter X, but doesn’t include T in its type parameter list. Because this arrangement prevents GenericBookParcel from passing type arguments to its base class Parcel, it fails to compile.

You can also define new type parameters for a derived class when you extend a generic base class. In the following example, class GenericBookParcel defines two type parameters X and T:

class Parcel<**T**> {}

class GenericBookParcel<**X, T**> extends Parcel<**T**> {}

Here’s another example, in which the derived class passes type arguments to its generic base class in its declaration:

class Parcel<**T**> {}

class GenericBookParcel<**X**> extends Parcel<**Book**> {} //**Type argument Book passed to base class Parcel**

EXAM TIP A *type argument* must be passed to the *type parameter* of a base class. You can do so while extending the base class or while instantiating the derived class.

NONGENERIC CLASS EXTENDING A GENERIC CLASS

You can extend a generic base class to define a non-generic base class. To do so, the derived class doesn’t define any type parameters but passes arguments to all type parameters of its generic base class. For example

class Parcel<T>{}

class NonGenericPhoneParcel extends Parcel<Phone> {}

In the preceding example, NonGenericPhoneParcel is a non-generic class that passes argument Phone to its base class Parcel<T>.

Watch out for exam questions that try to pass type arguments to a non-generic class. For class NonGenericPhoneParcel defined in the preceding example code, the following code won’t compile:

NonGenericPhoneParcel<String> var = new NonGenericPhoneParcel<>();

// **Won’t compile**

EXAM TIP You can’t pass type arguments to a non-generic class.

MULTIPLE TYPE PARAMETERS

The example of generic class Parcel used in this section defines one type parameter. A generic class with multiple type parameters takes the following form:

class ClassName <T1, T2, …, Tn> { /\* code \*/}

NONGENERIC CLASS IMPLEMENTING A GENERIC INTERFACE

class **MapLegendNonGeneric** implements **MyMap<String, Integer>** {

public void put(String s, Integer i) {}

public Integer get(String s) { return null; }

}

In the preceding example, MapLegendNonGeneric is a nongeneric class which implements generic interface MyMap. When implementing a generic interface, take note of the type parameters and how they are used in method declarations. The methods of an implementing class must implement or override all the interface methods. In the following example, class MapLegendNonGeneric won’t compile because it doesn’t override the abstract method get(String) in MyMap (the return type of get() is declared to be String, not Integer):

class MapLegendNonGeneric implements MyMap<String, Integer> {

public void put(String s, Integer i) {}

public **String** get(String s) { return null; }// **Won’t compile**

}

***Using generic methods***

If you are writing Generics method then you need to declare type parameters in method signature between method modifiers and return type as shown in below Java Generics example:

**public** **static** <T> T identical(T source){  
        **return** source;  
 }

Failing to declare <T> will result in compile time error. to know more read How to write Generics method in Java.

EXAM TIP For a generic method (defined in a nongeneric class or interface), its type parameter list is placed just after the access and non-access modifiers and before its return type.

EXAM TIP A non-generic class can implement a generic interface by replacing its type parameters with actual types.

GENERIC CLASS IMPLEMENTING A GENERIC INTERFACE

interface MyMap<K, V>{

void put(K key, V value);

V get(K key);

}

class MapLegendGeneric<K, V> implements MyMap<K, V> {

public void put(K key, V value) { }

public V get(K key) { return null; }

}

***Bounded type parameters***

Without a bounded type parameter (and explicit type casting), you can access only the members defined in the superclass of all classes—that is, class Object.

In the following example, the generic class Parcel won’t be able to access method getWeight() of class Gift:

abstract class Gift{

abstract double getWeight();

}

class Book extends Gift{

public double getWeight() {return 3.2;}

}

class Phone extends Gift{

public double getWeight() { return 1.1; }

}

class Parcel<T>{

private T t;

public void set(T t) {

this.t = t;

}

public void shipParcel() {

if (**t.getWeight()** > 10)// **Won’t compile; type of t is Object.**

System.out.println("Ship by courier ABC");

else

System.out.println("Ship by courier XYZ");

}

}

To access members of Gift in Parcel, you can limit the type of objects that can be passed to class Parcel (to Gift and its subclasses) by using bounded parameters.

class Parcel<**T extends Gift**>{//**Bounded type parameter**

private T t;

public void set(T t) {

this.t = t;

}

public void shipParcel() {

if (**t.getWeight()** > 10)// **Compiles; type of t is Gift.**

System.out.println("Ship by courier ABC");

else

System.out.println("Ship by courier XYZ");

}

}

Parcel<String> p = new Parcel<>();

The preceding code will not compile because the type argument String isn’t within bounds of type variable T.

EXAM TIP For a bounded type parameter, the bound can be a class, interface, or enum, but not an array or a primitive type. All cases use the keyword extends to specify the bound. If the bound is an interface, the implements keyword isn’t used.

***Using wildcards***

There are generally two kinds of wildcards in Generics, Bounded and unbounded. Bounded wildcards can be written in two ways to denote upper bound and lower bound. <?> is called unbounded wildcards because it can accept any Type while <? extends T> and <? super T> are bounded wildcards.

**<?>**

*"*?" denotes any unknown type, It can represent any Type at in code for. Use this wildcard if you are not sure about Type. for example, if you want to have an ArrayList which can work with any type than declare it as  "ArrayList<?> unknownList" and it can be assigned to any type of ArrayList as shown in following an example of generics in Java:

ArrayList<?> unknownList = new ArrayList<Number>();  
unknownList = new ArrayList<Float>();

**<? extends T>**

This is little restrictive than the previous one it will allow All Types which are either "T" or extends T means a subclass of T. for example List<? extends Number> can hold List<Number> or List<Integer>

ArrayList<? extends Number> numberList = new ArrayList<Number>();  
numberList = new ArrayList<Integer>();  
numberList = new ArrayList<Float>();

**<T super ?>**

This is just opposite of previous one, It will allow T and super classes of T, e.g. List<? super Integer>can hold List<Integer> or List<Number>.

ArrayList<? super Integer> numberList = new ArrayList<Number>();  
numberList = new ArrayList<Integer>();  
numberList = new ArrayList<Float>(); *//compilation error*

class Gift{}

class Book extends Gift{}

class Phone extends Gift{}

You can assign an object of class Book or Phone to a reference variable of type Gift:

Gift gift = new Book();

gift = new Phone();

But the following assignment isn’t valid:

List<Gift> wishList = new ArrayList<Book>();//**Won’t compile**

You can use a wildcard to get around this. In the following example, you can assign an ArrayList of *any* type to wishList:

List<?> wishList = new ArrayList<Book>();//**? refers to any type**

Because ? refers to an unknown type, wishList is a list of an unknown type. So it’s acceptable to *assign* a list of Book objects to it.

List<?> wishList = new ArrayList<Book>();

wishList.add(new Book());

**Wildcard with method:**

public static void wrapGift(**List<?> list**) {

for (**Object item : list**) {

System.out.println("GiftWrap - " + item);

}

}

public static void main(String args[]) {

List<Book> bookList = new ArrayList<Book>();

bookList.add(new Book("Oracle"));

bookList.add(new Book("Java"));

wrapGift(bookList);

List<String> stringList = new ArrayList<String>();

stringList.add("Paul");

stringList.add("Shreya");

wrapGift(stringList);

}

EXAM TIP When you use a wildcard to declare your variables or method parameters, you lose the functionality of adding objects to a collection. In this case, using the add method will result in compilation failure.

EXAM TIP In upper-bounded wildcards, the keyword extends is used for both a class and an interface.

Consider the following classes:

class Gift{}

class Book extends Gift{}

class Phone extends Gift{}

For a variable that uses the upper-bounded wildcard <? extends Gift>, the following assignments are valid:

List<? extends Gift> myList1 = new ArrayList<Gift>();

List<? extends Gift> myList2 = new ArrayList<Book>();

List<? extends Gift> myList3 = new ArrayList<Phone>();

upper-bounded wildcard in method parameters

public static void wrapGift(List<**? extends Gift**> list) {

for (Gift item : list) {

System.out.println("GiftWrap - " + item);

}

}

For the preceding method, you can pass to it List of Gift or objects that extend class Gift. If you try to pass it a list of any other object type, it won’t compile.

EXAM TIP In the preceding method wrapGift(), the loop variable item can be of type Gift or its subtype, Object.

EXAM TIP For collections defined using upper-bounded wildcards, you can’t add any objects. You can iterate and read values from such collections.

EXAM TIP You can use final classes in upper-bounded wildcards. Although class X extends String won’t compile, <? extends String> will compile successfully.

Lower-bounded wildcard <? super Gift>

class Gift{}

class Book extends Gift{}

class Phone extends Gift{}

List<? super Gift> myList1 = new ArrayList<Gift>();

List<? super Gift> myList2 = new ArrayList<Object>();

List<? super Gift> myList3 = new ArrayList<Phone>();//**Won’t compile; gift doesn’t extend Phone.**

List<? super Phone> myList4 = new ArrayList<Gift>();//**Valid; Phone extends Gift.**

So, what can you read from and add to collection objects defined using lower-bounded wildcards? Here’s an example:

List<**? super Gift**> list = new ArrayList<**Gift**>();

list.add(**new Gift()**);

list.add(**new Book()**);

list.add(**new Phone()**);

// **Can add instances of Gift or its subclasses to List<? super Gift>.**

list.add(new Object());//**Won’t compile**

for (**Object obj** : list) System.out.println(obj);

EXAM TIP When a generic class is compiled, you don’t get multiple versions of the compiled class files. A generic class gets compiled into a single class file, erasing the type information during the compilation process.

The compiler erases the type information by replacing all type parameters in generic types with Object (for unbounded parameter types) or their bounds (for bounded parameter types). The compiler might insert type casts to preserve type safety and generate bridge methods to preserve polymorphism in extended generic types.

When generics were introduced with Java 5, it was mandatory to include the type arguments to instantiate a generic class.

Parcel**<String>** parcel = new Parcel**<String>**();//**Type arguments included to invoke constructor of generic class Parcel.**

But with Java 7, you can drop the type arguments required to invoke the constructor of a generic class and use an empty set of type arguments, <>(Daimond):

Parcel**<String>** parcel = new Parcel**<>**();

Parcel<String> parcel = new Parcel();//**Won’t compile**

***Mixing reference variables and objects of raw and generic types***

When you use a reference variable of a raw type, you lose the type information.

class Parcel<T> {

private T t;

public void set(T t) {

this.t = t;

}

public T get() {

return t;

}

}

Parcel parcel = new Parcel<Phone>();

parcel.set("harry");//**Warning: Because you lose type information when you use variable of raw type, you can pass String object to set(), instead of Phone object.**

Phone phone = parcel.get();//**Won’t compile; with reference variable of raw type**

**get() method returns Object it doesn’t know Phone type.**

Phone phone = (Phone)parcel.get();//**Compile successfully.**

EXAM TIP When you mix raw with generic types, you might get a compiler warning or error, or a runtime exception.

List List = new ArrayList<String>();

list.add(new String("Shreya"));// **Warning**

list.add(new Integer(1));// **Warning**

list.add(new Object());//**Warning**

String value = list.get(0);// **Won’t compile**

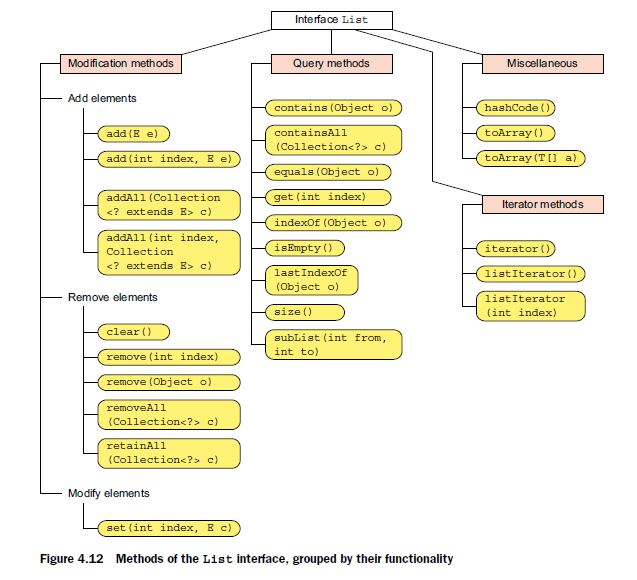
Now, let’s try to assign a raw type to a parameterized type:

Parcel<Phone> parcel = new Parcel();//**compiler warnings**

parcel.set(new Phone());//**No compiler warnings**

//parcel.set(new String());//**Won’t compile**

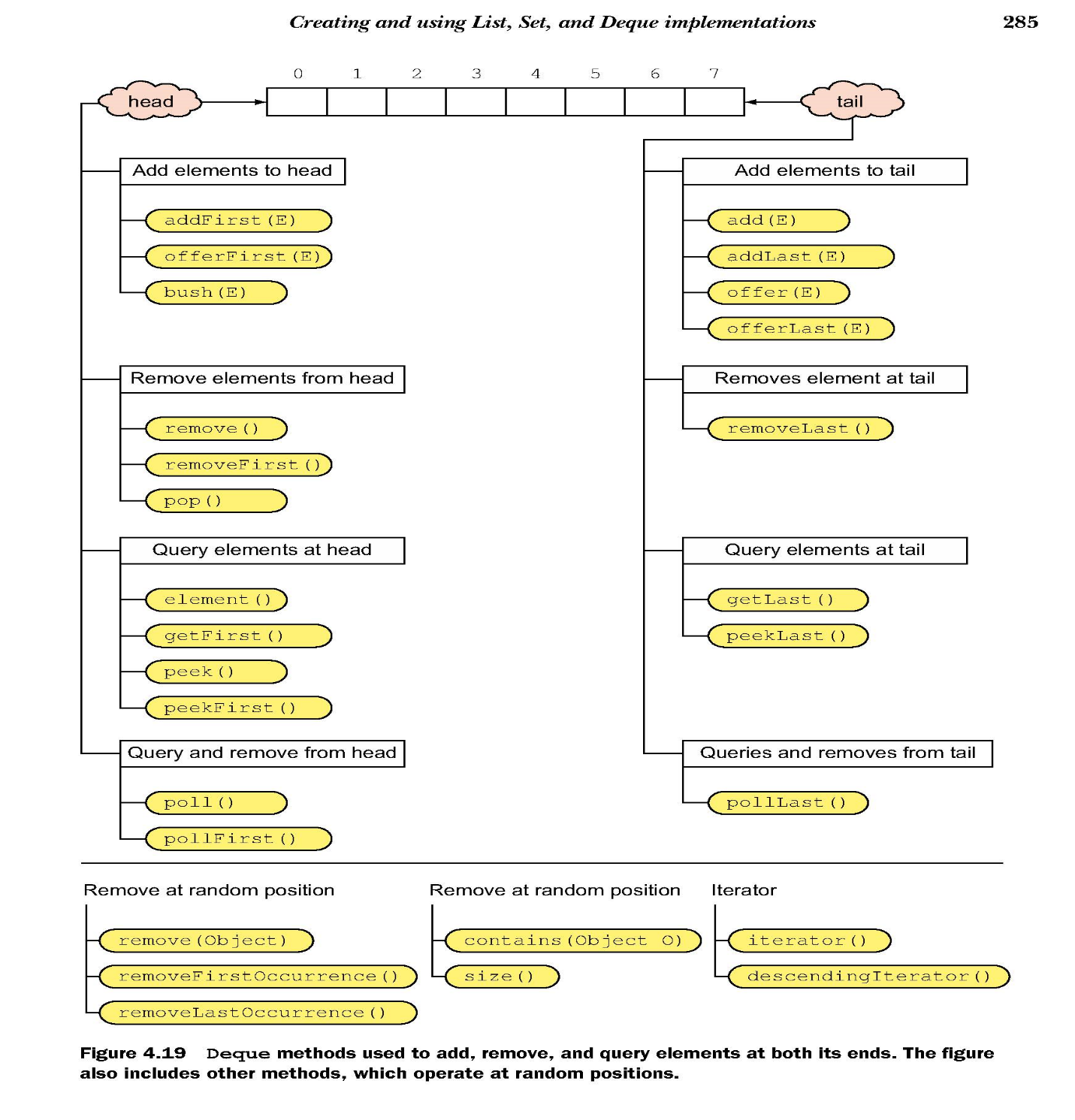
Phone phone = parcel.get();//**Compiles successfully**

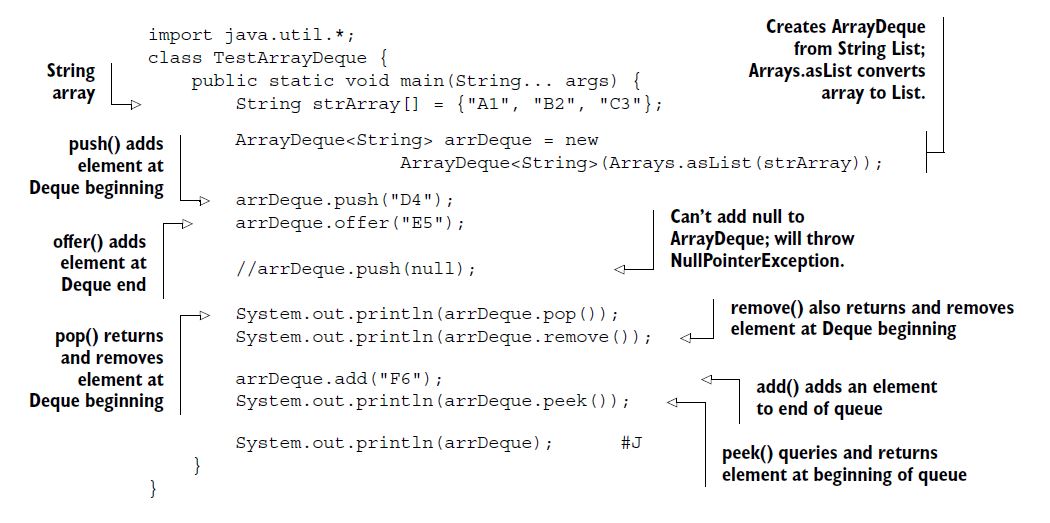


EXAM TIP To remove an element, an ArrayList first searches through its elements to find an element that can be considered equal to the target element. It does so by calling method equals() on the target object and its own objects, one by one. If a matching element is found, remove(Object)removes the *first* occurrence of the match found.

EXAM TIP If you’re adding instances of a user-defined class as elements to an ArrayList, override its method equals() or else its methods contains() or remove() might not behave as expected.

EXAM TIP The ArrayList methods clear(), remove(), and removeAll() offer different functionalities. clear() removes all the elements from an ArrayList. remove(Object) removes the first occurrence of the specified element, and remove(int) removes the element at the specified position. removeAll() removes from an ArrayList all of its elements that are contained in the specified collection.





Here’s the output of the preceding code:

D4

A1

B2

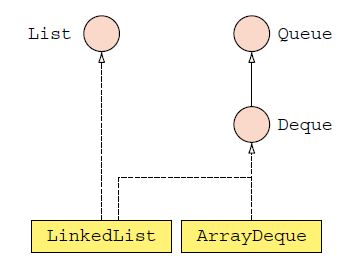
[B2, C3, E5, F6]

You also need to take note of methods like peek(), which only queries Deque; remove(), which removes elements from Deque; and poll(), which queries and removes an element from Deque. Method poll() queries and removes, and method remove() just removes. Method poll() returns null when Deque is empty and remove() throws a runtime exception.

EXAM TIP All the insertion methods (add(), addFirst(), addLast(), offer(), offerFirst(), offerLast(), and push()) throw a NullPointer- Exception if you try to insert a null element into an ArrayDeque.

LINKEDLIST

Class LinkedList implements both the List and Deque interfaces.



LinkedList<String> list = new LinkedList<String>();

list.offer("Java");

list.push("e");

list.add(1, "Guru");

System.out.println(list.remove("e"));//true

Iterator<String> it = list.iterator();

while(it.hasNext()) System.out.println(it.next());//Guru, Java

EXAM TIP Because a LinkedList implements List, Queue, and Deque, it implements methods from all these interfaces.

EXAM TIP Because a LinkedList implements List, Queue, and Deque, it implements methods from all these interfaces.

EXAM TIP Watch out for questions that add null to a HashSet. A Hash- Set allows storing of only one null element. All subsequent calls to storing null values are ignored.

EXAM TIP In the absence of passing a Comparator instance to a TreeSet constructor, the objects that you add to a TreeSet must implement Comparable otherwise you will get ClassCastException. In the preceding example, String (which implements Comparable) objects are added to the TreeSet. Watch out for storing objects of wrapper classes, Enum and File in a TreeSet; they all implement Comparable. The natural order of enum constants is the order in which they’re declared. StringBuffer and StringBuilder don’t implement Comparable.

Map<String, Double> salaryMap = new HashMap<>();

salaryMap.put("Paul", 8888.8);

salaryMap.put("Shreya", 99999.9);

Map<String, Object> copySalaryMap = new HashMap(salaryMap);

Set<String> keys = copySalaryMap.keySet();

for (String k : keys)

System.out.println(k);

salaryMap.remove("Paul");

keys = copySalaryMap.keySet();

for (String k : keys)

System.out.println(k);//**Still outputs two key values.**

EXAM TIP You can create a HashMap by passing its constructor another Map object. Additions of new key-value pairs or deletions of existing key-value pairs in the Map object passed to the constructor aren’t reflected in the newly created HashMap.

Map<String, Double> salaryMap = new HashMap<>();

Map<Object, String> copySalaryMap = new HashMap<>(salaryMap);// **Won’t compile**

EXAM TIP The String class and all the wrapper classes override their hashCode() and equals() methods. So they can be correctly used as keys in a HashMap.

EXAM TIP When objects of a class that only overrides method equals() and not method hashCode() are used as keys in a HashMap, contains- Key() will always return false. But containsValue() returns true because it uses only equals() method and not method hashCode() to determine the equality of HashMap values.

Map<Integer, String> map = new HashMap<>();

map.put(1, "Shreya");

map.put(11, "Paul");

Map<Integer, String> anotherMap = new HashMap<>();

anotherMap.put(1, "Harry");

anotherMap.putAll(map);

System.out.println(anotherMap);

Output:

{1=Shreya, 11=Paul}

EXAM TIP Method values() returns a Collection object, method key- Set() returns a Set object, and method entrySet() returns a Map.Entry object.

HashMap and HashSet uses methods hash-Code() and equals() of its key to add, remove, or query it. But TreeMap performs all key comparisons by using method compareTo() or compare() of its keys. Two keys are considered equal by a TreeMap if the key’s method compareTo() or compare() considers them equal.

When you create a TreeMap object, you should specify how its keys should be ordered. A key might define its natural ordering by implementing the Comparable interface. If it doesn’t you should pass a Comparator object to specify the key’s sort order.

enum IceCream {STRAWBERRY, CHOCOLATE, WALNUT};

Map<IceCream, String> flavorMap = new TreeMap<>();

flavorMap.put(IceCream.CHOCOLATE, "Paul");

flavorMap.put(IceCream.STRAWBERRY, "Shreya");

flavorMap.put(IceCream.WALNUT, "Muni");

for (String s : flavorMap.values())

System.out.println(s);

The output of the preceding code is:

Shreya

Paul

Muni

EXAM TIP The natural order of enum elements is the sequence in which they’re defined. The set of *values* that you retrieve from a TreeMap is sorted on its *keys* and not on its *values*.

EXAM TIP You can create a TreeMap without passing it a Comparator object or without using keys that implement the Comparable interface. But adding a key-value pair to such a TreeMap will throw a runtime exception, ClassCastException.

Class TreeMap implements the SortedMap interface. Watch out for similar code on the exam that tries to instantiate a SortedMap. It won’t compile. For example

Map<String, String> map = new SortedMap<String, String>();//**Won’t compile**

COMPARING KEYS: TREEMAP VERSUS HASHMAP

Unlike a HashMap, a TreeMap uses method compare() or compareTo() to determine the equality of its keys. In the following example, a TreeMap can access the value associated with a key, even though its key doesn’t override its method equals() or hashCode():

class Flavor implements Comparable<Flavor> {

String name;

Flavor(String name) {

this.name = name;

}

public int compareTo(Flavor f) {

return this.name.compareTo(f.name);

}

}

class CreateTreeMap {

public static void main(String args[]) {

Map<Flavor, String> flavorMap = new TreeMap<>();

flavorMap.put(new Flavor("Chocolate"), "Paul");

flavorMap.put(new Flavor("Apple"), "Harry");

System.out.println(flavorMap.get(new Flavor("Apple")));// **Prints**

**“Harry”**

}

}

----------------------------------------------------------------

class Person implements Comparable<Person> {

String name;

int age;

Person (String name, int age) {

this.name = name;

this.age = age;

}

**public int compareTo(Person person) {**

**return 0;**

**}**

public String toString() {

return name;

}

}

class TestComparable {

public static void main(String args[]) {

TreeSet<Person> set = new TreeSet<>();

Person p1 = new Person("Shreya", 12);

Person p2 = new Person("Harry", 40);

Person p3 = new Person("Paul", 30);

set.add(p1);

set.add(p2);

set.add(p3);

Iterator<Person> iterator = set.iterator();

while(iterator.hasNext()) {

System.out.println(iterator.next());

}

}

}

Classes like TreeSet and TreeMap store their elements in a sorted order. Before set adds the second element, p2, it compares it to the existing element, p1. Because p1.compareTo(p2) returns 0, set doesn’t add the *duplicate element* and returns false. The same steps are repeated when set tries to add p3. At the end, only one element, p1, is added to set.

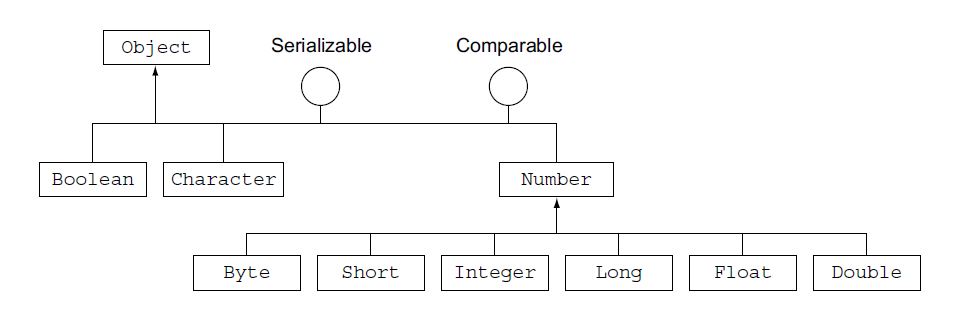
The Comparator interface is used to define the *sort order* of a collection of objects, without requiring them to implement this interface. This interface defines methods compare() and equals(). You can pass Comparator to sort methods like Arrays.sort and Collections.sort. It’s also passed to collection classes like TreeSet and TreeMap that require ordered elements.

The Comparator interface is used to specify the sort order for classes that

* Don’t define a natural sort order
* Need to work with an alternate sort order
* Don’t allow modification to their source code so that natural ordering can be added to them

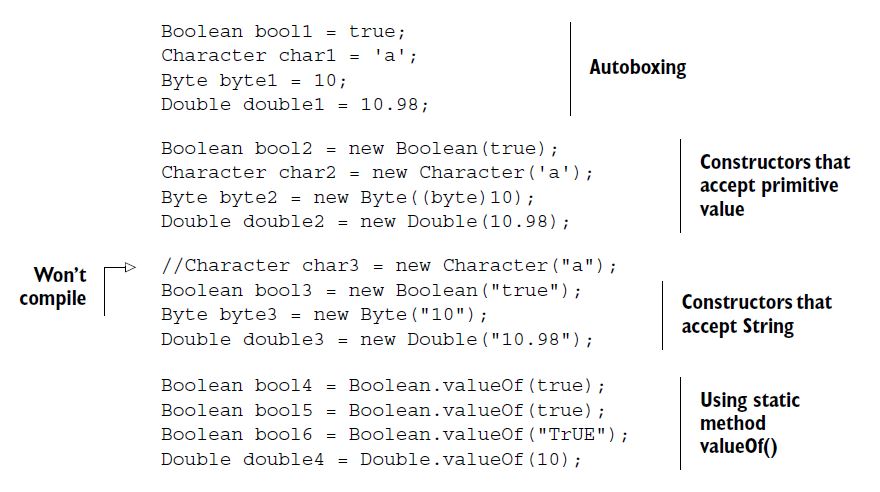
EXAM TIP Once Elements are sorted, new elements are added to its end.

All the wrapper classes are immutable. All the numeric wrapper classes extend the class java.lang.Number. Classes Boolean and Character directly extend class Object. All the wrapper classes implement the interfaces java.io.Serializable and java.lang.Comparable. All these classes can be serialized to a stream, and their objects define a natural sort order.



You can create objects of all the wrapper classes in multiple ways:

* *Assignment*—By assigning a primitive to a wrapper class variable
* *Constructor*—By using wrapper class constructors
* *Static methods*—By calling the static method of wrapper classes, like valueOf()



You can create objects of the rest of the wrapper classes (Short, Integer, Long, and Float) in a similar manner. All the wrapper classes define constructors to create an object using a corresponding primitive value or as a String.

Another interesting point to note is that neither of these classes defines a default no-argument constructor. Because wrapper classes are immutable, it doesn’t make sense to initialize the wrapper objects with the default primitive values if they can’t be modified later.

EXAM TIP All wrapper classes (except Character) define a constructor that accepts a String argument representing the primitive value that needs to be wrapped. Watch out for exam questions that include a call to

a no-argument constructor of a wrapper class. None of these classes defines a no-argument constructor.

Primitive to object ----- valueOf();

Object to Primitive ---- XXXValue();

String to Primitive ----- parseXXX();

***Between Using Method valueOf() And Constructors Of Wrapper Classes***

Wrapper classes Character, Byte, Short, Integer, and Long cache objects with values in the range of –128 to 127. These classes define inner static classes that store objects for the primitive values –128 to 127 in an array. If you request an object of any of these classes, from this range, method valueOf() returns a reference to a predefined object; otherwise, it creates a new object and returns its reference:

Long var1 = Long.valueOf(123);

Long var2 = Long.valueOf("123");

System.out.println(var1 == var2);//**true**

Long var3 = Long.valueOf (223);

Long var4 = Long.valueOf (223);

System.out.println (var3 == var4);//**false**

EXAM TIP When arranged in natural sort order, false precedes true.

public int increment(Integer obj) {

return ++obj;

}

Because the Java compiler would call obj.intValue() to get obj’s int value, passing null to method increment() will throw a NullPointerException.

EXAM TIP Unboxing a wrapper reference variable, which refers to null, will throw a NullPointerException.

The hashCode value is used to test for object inequality. If two objects return different hashCode values, they can never be equal. But if your objects return the same hashCode values, they can be equal (if their equals() returns true).

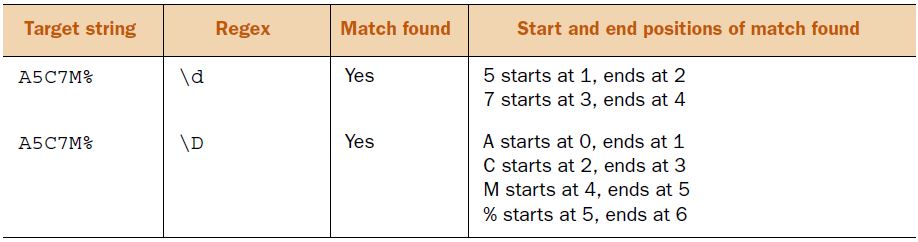
EXAM TIP If a matched string occupies index positions 1, 2, and 3 in a target string, method end() of class Matcher returns the value 4 for the corresponding call on end(). You can expect trick questions on this returned value on the exam.

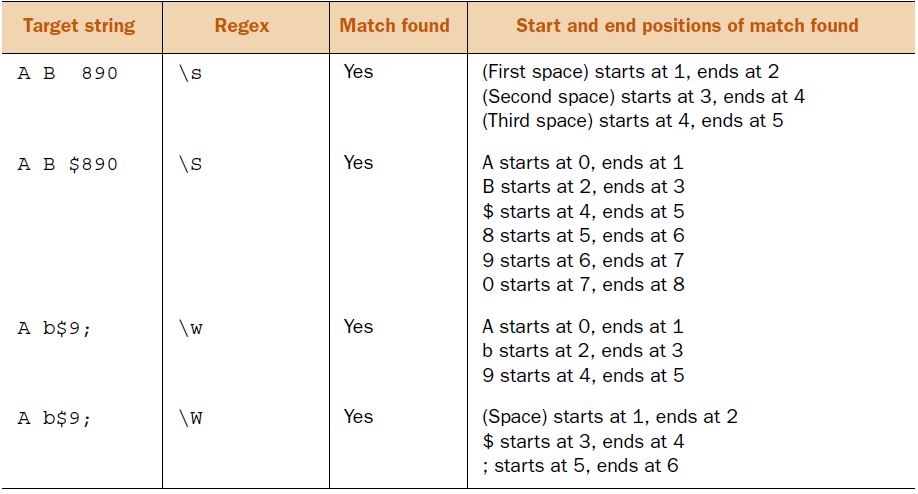
EXAM TIP If the Java Runtime engine determines that a pattern is invalid, it throws the runtime exception *PatternSyntaxException*. On the exam, when you see a question on the possible output of a string processing code, examine the regex pattern for invalid values.

|  |  |  |
| --- | --- | --- |
| Class type | Regex pattern | Description |
| Simple | [agfd] | Match exactly one from a, g, f, or d |
| Range | [a-f0-7] | Match exactly one from the range a to f (both inclusive) or 0  to 7 (both inclusive) |
| Negation | [^123k-m] | Match exactly one character that is not 1, 2, or 3 or from the  range k to m (both inclusive) |

*Predefined character classes*

|  |  |
| --- | --- |
| Character class | Description |
| . (Dot) | Any character (may or may not match line terminators) |
| \d | A digit: [0-9] |
| \D | A non-digit: [^0-9] |
| \s | A whitespace character: [ space, \t (tab), \n (new line), \x0B (end of line), \f (form feed), \r (carriage)] |
| \S | A non-whitespace character: [^\s] |
| \w | A word character: [a-zA-Z\_0-9] |
| \W | A non-word character: [^\w] |





String targetString = "A b$9;";

String regex = "\\W";

Pattern pattern = Pattern.compile(regex);

Matcher matcher = pattern.matcher(targetString);

String replacedStr = matcher.replaceAll("");

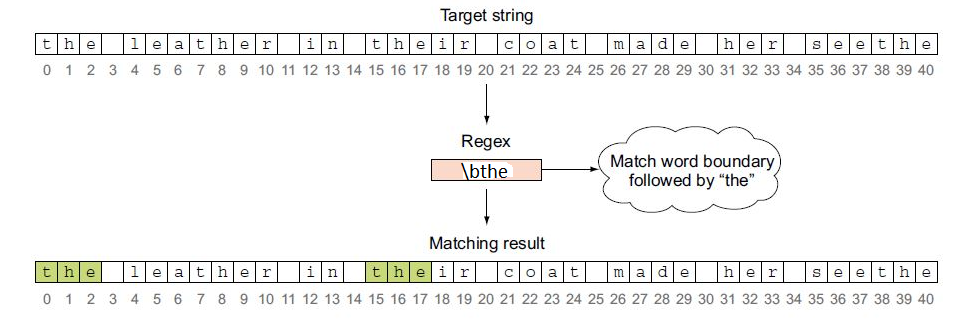
System.out.println(replacedStr);//**Ab9**

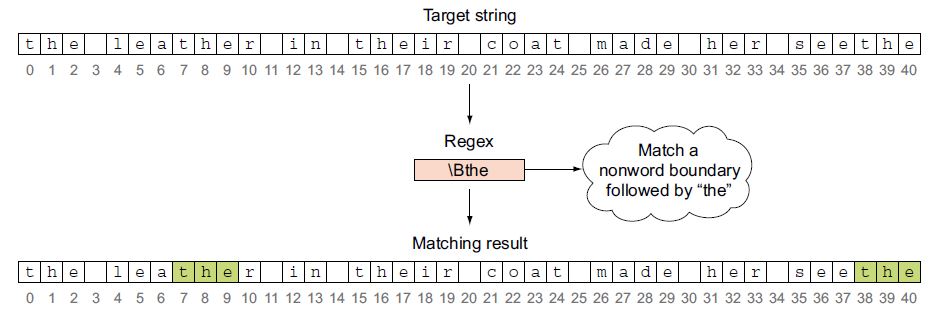
EXAM TIP Because String objects are immutable, calling replaceAll() won’t change the contents of String referred to by the variable target- String in the preceding code example. replaceAll() creates and returns a new String object with the replaced values. Watch out for questions based on it on the exam.

*Matching boundaries*

Say you want to find all occurrences of the word *the* in your book. To do that, you’d need to search for the text *the*. Well, the same is true when you want to match *the*, which can be part of another word, for example, *their, leather*, or *seethe*. So you need a way to limit your searches to the start or end of a word. *Matching boundaries* can help you with this. You can match boundaries including the start of a line, a word, a non-word, or the end of a line by using regex patterns. Even though the boundary constructs ^ (beginning of line) and $ (end of line) aren’t explicitly included in the exam objectives, you might see them in answer options that are incorrect.

|  |  |
| --- | --- |
| Boundary construct | Description |
| \b | A word boundary |
| \B | A non-word boundary |
| ^ | Beginning of a line |
| $ | End of a line |





\bthe means a word which will start with the and any characters after that.

\Bthe means a word which contains the in any place but not at beginning.

the\b means a word that ends with the.

Consider the following string literal value:

String targetString = "The leather in their coat made her seethe";

Which of these options correctly defines a regex pattern for searching the literal string for *the* at either the beginning or end of a word, but not in its middle?

1. String regex = "\\Bthe\\B";//**which isn't placed at the start or end of a word.**
2. String regex = "\\bthe\\B";//**start with “the” but don't end with the.**
3. String regex = "\\Bthe\\b";//**don't start with “the” but end with “the”.**
4. String regex = "\\bthe|the\\b";//**starts with “the” or ends with “the”.**

0[xX][0-9a-fA-F]

The preceding expression could be stated:

Find a set of characters in which the first character is a "0", the second character is either an "x" or an "X", and the third character is a digit from "0" to "9", a letter from "a" to "f", or an uppercase letter from "A" to "F".

*The Predefined Dot*

In addition to the \s, \d, and \w metacharacters that we discussed, you have to understand the "." (dot) metacharacter. When you see this character in a regex expression, it means "any character can serve here." For instance, the following source and pattern:

source: "ac abc a c"

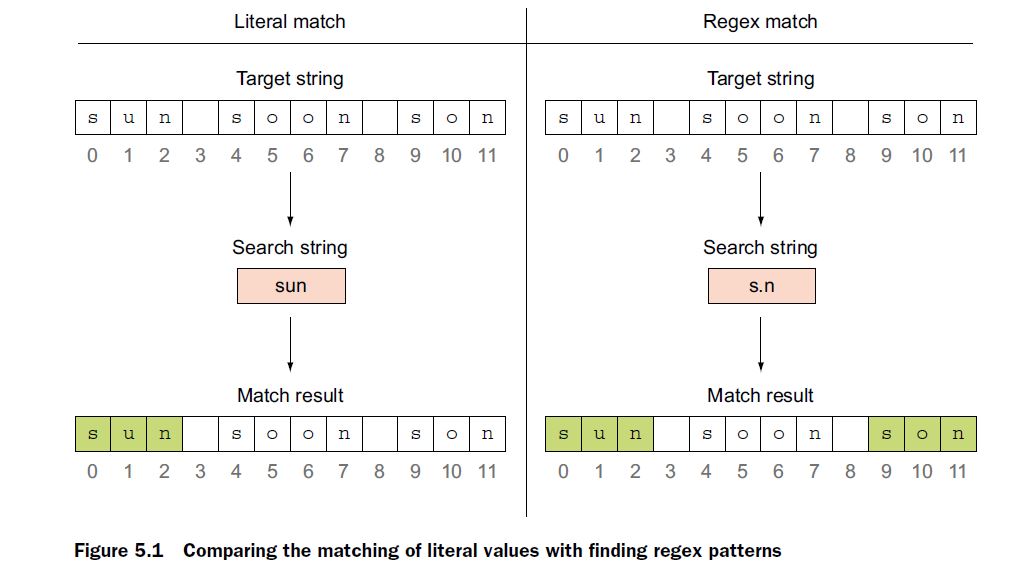
pattern: a.c

will produce the output

3 abc

7 a c

The "." was able to match both the "b" and the " " in the source data.



*Quantifiers*

|  |  |
| --- | --- |
| Quantifier (Greedy) | Description |
| X? | Matching X, once or not at all (0 or 1) |
| X\* | Matching X, zero or more times (0 or many) |
| X+ | Matching X, one or more times (1 or more) |

***Search string*:** It can be written as August or Aug

***Regex*:** Aug(ust)? (After “Aug”, “ust” accurse 0 or 1’s.)

**Matches**: August or Aug

***Search string:*** A ball can fall in a mall with all

***Regex***: [bmf]?all (b or m or f accurse 0 or 1’s)

**Matches:** ball, mall, fall, or all

***Search string:*** food, fod, fooodder, fd

***Regex***: fo\*d (o accurse 0 or more)

**Matches:** *food*, *fod*, *foood, fd*

***Search string*:** b234a A6Z abc

***Regex*:** [A-Za-z]\d\*[A-Za-z] (second letter is digit, digit accurse 0 or more times)

**Matches:** b234a, A6Z, ab

It matches ab because \d\* asked it to look for zero or more occurrences of digits. Because ab has zero digits between *a* and *b*, it’s matched. It won’t match bc because b was already consumed for matching ab.

Now, what if you need to modify the preceding example, to limit the digits that appear between the letters to a range of 1 to 5? This is simple: just replace \d\* with [1-5].

***Search string:*** b234a A6Z abc

***Regex*:** [A-Za-z][1-5]\*[A-Za-z]

**Matches:** b234a, ab

***Search string:*** food, fod, fooodder, fd

***Regex:*** fo+d

**Matches:** *food*, *fod*, *food*

***Search string:*** b234a A6Z abc

***Regex*:** [A-Za-z]\d+[A-Za-z]

**Matches:** b234a A6Z

***Search string*:** b234a A6Z abc

***Regex*:** [A-Za-z][1-5]+[A-Za-z]

**Matches:** b234a

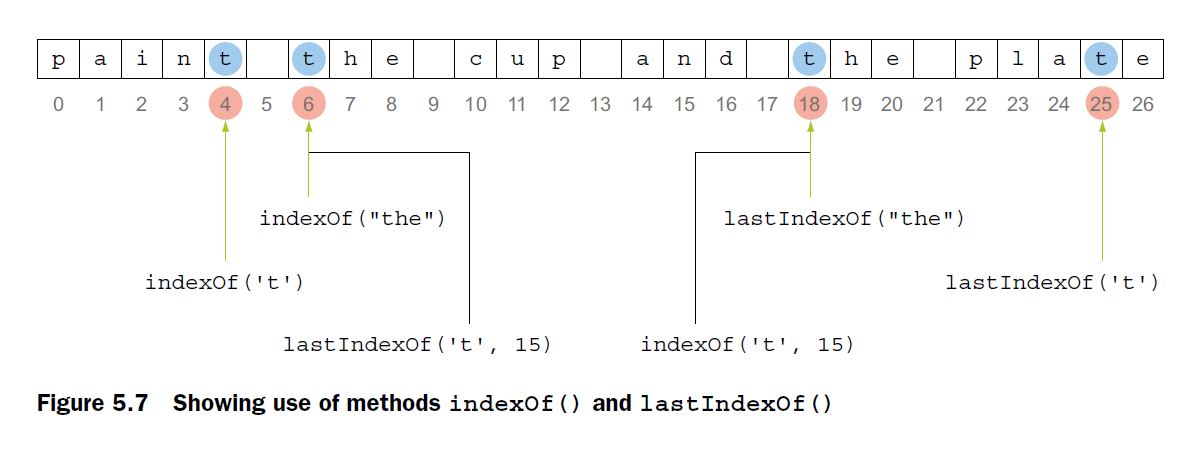
***Search string*:** The1new22Programmer exam6

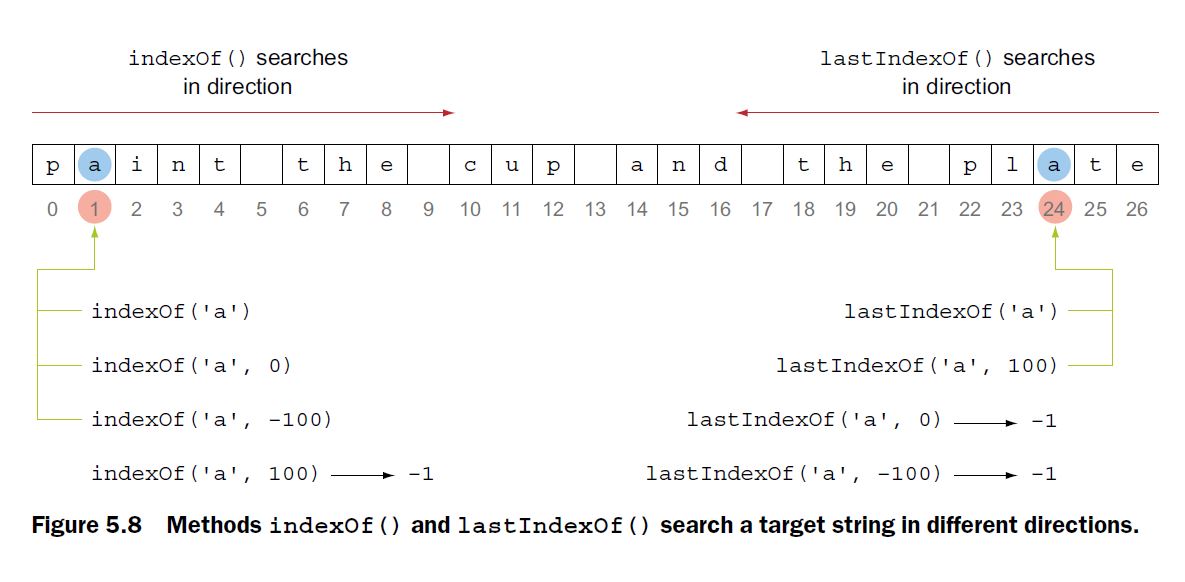
***Regex*:** [\\sA-Za-z]+

**Matches:** The, new, Programmer exam

NOTE The use of StringTokenizer is discouraged in new code. This legacy class is retained for backward compatibility. Use classes from the java.util.regex package or String.split() to get the functionality of StringTokenizer.

*Searching strings:*





EXAM TIP Methods indexOf() and lastIndexOf() don’t throw a compilation error or runtime exception if the search position is negative or greater than the length of the string. If no match is found, they return -1.

METHOD CONTAINS()

Method contains() searches for exact matches in a string and returns true if a match is found, false otherwise. Because contains() accepts a method parameter of type CharSequence, you can pass to it both a String or a StringBuilder object:

String sentence = "paint the cup and the plate";

StringBuilder sb = new StringBuilder("the");

String str = "the";

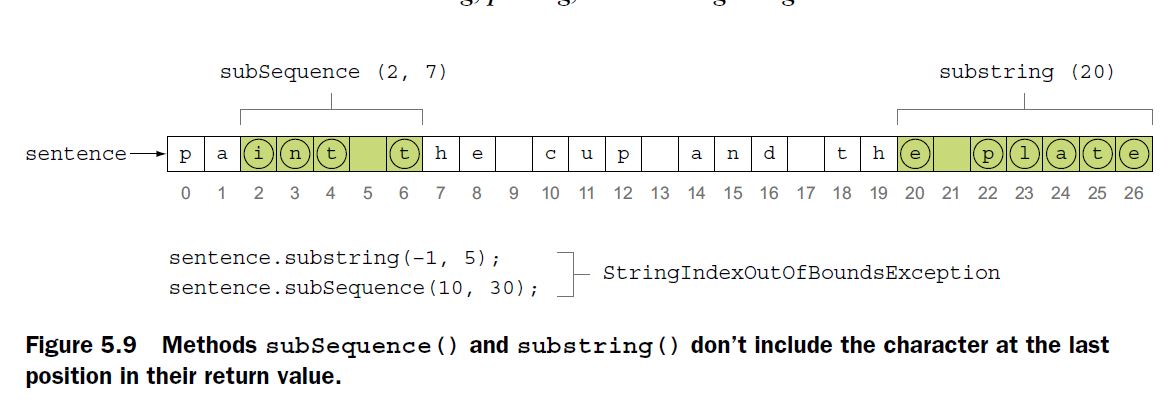
System.out.println(sentence.contains(sb));

System.out.println(sentence.contains(str));

EXAM TIP To remember the return types of methods subSequence() and substring() on the exam, just remember that the names of these methods can be used to determine their return type. Method subSequence() returns CharSequence, and method substring() returns String.

CharSequence subSequence(int beginIndex, int endIndex)

String substring(int beginIndex)



EXAM TIP

Unlike indexOf() and lastIndexOf() methods, subSequence() and substring() methods throw the runtime exception StringIndexOutOfBoundsException for invalid start and end positions.

replace(char old, char new)

replace(CharSequence old,CharSequence new)

replaceAll(String regex, String replacement)

replaceFirst(String regex, String replacement)

EXAM TIP Unlike replace(), replaceAll() doesn’t accept method parameters of type CharSequence. Watch out for the passing of objects of class StringBuilder to replaceAll().

EXAM TIP Lexicographically, a lowercase letter is greater than its equivalent uppercase letter.

METHOD CONTAINS()

Method contains() searches for exact matches in a string and returns true if a match is found, false otherwise. Because contains() accepts a method parameter of type CharSequence, you can pass to it both a String or a StringBuilder object.

*Formatting strings*

Format strings using the formatting parameters %b, %c, %d, %f and %s in format strings. you can use class PrintStream and its overloaded methods format() and printf(), which use the default or specified Locale, format strings, and arguments:

format(String format, Object... args)

format(Locale l,String format, Object... args)

printf(String format, Object... args)

printf(Locale l, String format, Object... args)

Behind the scenes, method printf() simply calls method format().

EXAM TIP System.out.println() can’t write formatted strings to the standard output; System.out.format() and System.out.printf() can.

Here’s a quick example to write a formatted string to the standard output and to a file:

String name = "Shreya";

Formatter formatter = new Formatter(new File("data.txt"));

formatter.format("My name is **%s**", name);

formatter.flush();

System.out.printf("My name is %s", name);

%[argument\_index$][flags][width][.precision]conversion\_char

System.out.format("Name %b", name, age, isShort);//**Ignores extra variables**

System.out.printf("Name %b, age %b", name);//**Throws runtime exception**

System.out.format("\nName defined %10b.", name);

System.out.format("\nName defined %.1b.", name);

System.out.format("\nName defined %-10b.", name);

Here’s the output of the preceding code:

Name defined true.

Name defined t.

Name defined true .

EXAM TIP You can pass **any type of primitive variable** or **object reference** to %b.

EXAM TIP You can pass **only literals and variables** that can be converted to a Unicode character (char, byte, short, int, Character, Byte, Short, and Integer) to the %c specifier. Passing variables of type boolean, long, float, Boolean, Long, Float, or any other class will throw IllegalFormatConversionException.

EXAM TIP By default, %f prints six digits after the decimal. You can pass **literal values or variables** of type float, double, Float, and Double to the format specifier %f. But can’t pass int variables or literal values with %f. The following code will throw an IllegalFormatConversionException runtime exception.

EXAM TIP You can pass **literal values or variables** of type byte, short, int, long, Byte, Short, Integer, or Long to the %d format specifier. The code throws runtime exceptions for all other types of values.

EXAM TIP You can pass any type of **primitive variable or object reference** to the format specifier %s.

EXAM TIP You can specify that the argument\_index change the arguments used at a particular location.

*Exceptions and assertions*

*Using a method that throws a checked exception*

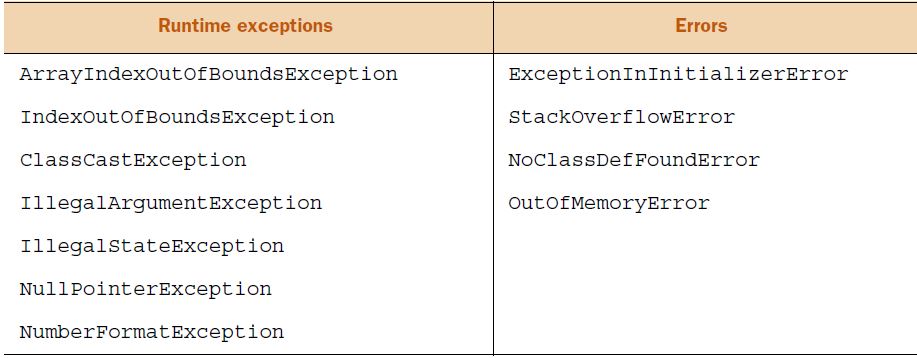
To use a method that throws a *checked exception*, you must do one of the following:

* *Handle the exception*—Enclose the code within a try block and *catch* the thrown exception.
* *Declare it to be thrown*—Declare the exception to be thrown by using the throws clause.
* *Handle and declare*—Implement both of the preceding options together.

*Using a method that throws runtime exceptions or errors*

While creating a method that throws a runtime exception or error, including the exception or error name in the throws clause isn’t required. A method that throws a runtime exception or error isn’t subject to the handle-or-declare rule.

EXAM TIP Adding runtime exceptions or errors to a method’s declaration isn’t required.



Method can throw a subclass of checked exception mentioned in its throws clause, not its superclass

class DemoThrowsException {

public void readFile(String file) **throws FileNotFoundException** {

boolean found = findFile(file);

if (!found)

**throw new IOException("Missing file");**

//**Won’t compile; can’t throw object of superclass of checked exception mentioned in throws clause.**

else {

//code to read file

}

}

boolean findFile(String file) {

//code to return true if file can be located

}

}

EXAM TIP An overriding method can throw any error or runtime exception, irrespective of whether they’re thrown by the overridden method or not.

Method can handle the exception and still declare it to be thrown

class DemoThrowsException {

public void readFile(String file) throws FileNotFoundException {

//..code

}

**void useReadFile(String name) throws FileNotFoundException {**

try {

readFile(name);

}

**catch (FileNotFoundException e) {**

//..code

**throw e;**

}

}

}

Method that declares a checked exception to be thrown might not actually throw it

class DemoThrowsException {

public void readFile(String file) throws FileNotFoundException {

// **Compiles successfully even if readFile() doesn’t throw FileNotFoundException.**

System.out.println("readFile:" + file);

}

}

You can rethrow exceptions with more-inclusive type checking

Starting with Java 7, the variable type that you use to rethrow an exception can bemore generic in the catch block:

class GenericVariableTypeToRethrowException {

public static void main(String args[])

throws **IOException**, **SQLException** {

String source = "DBMS";

try {

if (source.equals("DBMS"))

throw new SQLException();

else

throw new IOException();

}

catch (**Exception e**) {

**throw e**;

}

}

}

In the preceding code, the try block can throw two types of checked exceptions: SQLException or IOException. But the type of the variable e in the catch block is Exception, which is a superclass of SQLException and IOException. Prior to Java 7, this code would fail to compile because main() is trying to throw an object of Exception from its catch block, when its method declaration states that it can throw an IOException or SQLException. (For checked exceptions, a method can’t throw a superclass of the exception included in its declaration.) With Java 7, the preceding code compiles successfully, because the compiler can determine that the type of the checked exception received by the catch block would always be either IOException or SQLException. So it’s okay to throw it, even though the type of the variable e is Exception.

Do you think the code will compile successfully if instead of rethrowing the exception in the catch block, you create a new object of class Exception and throw it? No.

catch (**Exception e**) {

**throw new Exception();**//**Won’t compile; catch**

}

EXAM TIP With Java 7, you can rethrow exceptions with more inclusive type checking.

Class ThrowExceptions compiles successfully, even though its methods don’t include the code that might throw these exceptions:

class ThrowExceptions {

void method1() throws Error {}

void method2() throws Exception {}

void method3() throws Throwable {}

void method4() throws RuntimeException {}

void method5() throws FileNotFoundException {}

}

Though a try block can define a handler for unchecked exceptions not thrown by it, it can’t do so for checked exceptions (other than Exception):

class HandleExceptions {

void method6() {

try {}

catch (Error e) {}

}

void method7() {

try {}

catch (Exception e) {}

}

void method8() {

try {}

catch (Throwable e) {}

}

void method9() {

try {}

catch (RuntimeException e) {}

}

void method10() {

try {}

catch (FileNotFoundException e) {}

}

}

In the preceding code, method6(), method7(), method8(), and method9() compile even though their try blocks don’t define code to throw the exception being handled by their catch blocks. But method10() won’t compile.

EXAM TIP A method can declare to throw any type of exception—checked or unchecked—even if it doesn’t. But a try block can’t define a catch block for a checked exception (other than Exception) if the try block doesn’t throw that checked exception or uses a method that declares to throw that checked exception.

***Creating a custom checked exception***

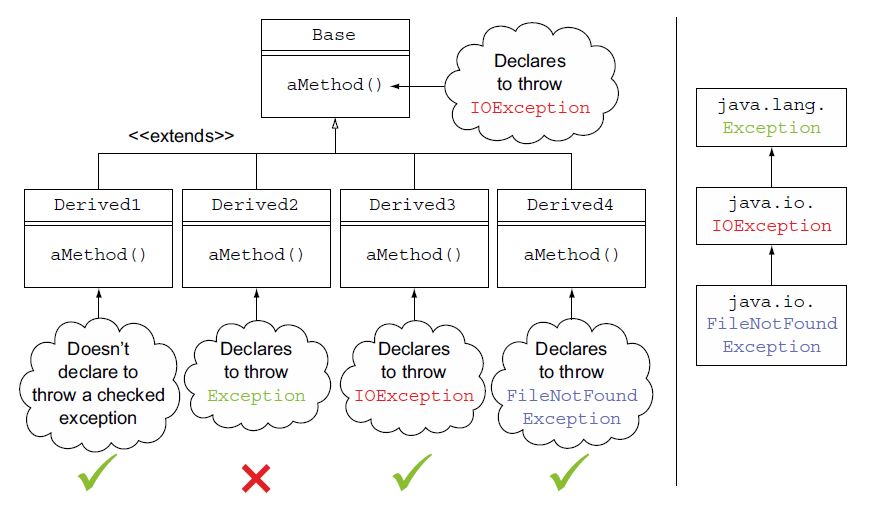
even though you can extend class java.lang.Throwable to create your own exceptions, it isn’t recommended. Class Throwable is the superclass of classes java.lang.Error and java.lang.Exception. The exception handler for this class will *catch* all types of errors and exceptions! For example, say an OutOfMemoryError brings the JVM into an unstable state. Catching it as a subclass of Error or Throwable would be undesirable and potentially dangerous. Obviously, you may not want this behavior.

EXAM TIP Don’t extend class java.lang.Throwable to create your own exceptions, even though you can. Class Throwable is the superclass of classes java.lang.Error and java.lang.Exception. The exception handler for this class will *catch* all types of errors and exceptions.

To create custom Error classes, you can subclass java.lang.Error by extending it. But Error classes represent serious exceptional conditions, which shouldn’t be thrown programmatically.

***Overriding methods that throw exceptions***

If an overridden method declares to throw a ***checked*** exception, the overriding method can choose to declare to not throw any checked exception, throw the same exception, or throw a more specific exception. The overriding method can’t declare to throw a more generic checked exception.



EXAM TIP Method overriding rules apply *only* to checked exceptions. They don’t apply to runtime exceptions or errors. Beware: you’re likely to be tested on this difference on the exam.

Rule 1: If a base class method doesn’t declare to throw a checked exception, an overriding method in the derived class can’t declare to throw a checked exception

Examine the following code:

class Base {

public void aMethod(){}

public void bMethod(){}

}

class Derived extends Base {

public void aMethod() throws Exception{}//**fails to compile.**

public void bMethod() throws RuntimeException{}//**compiles successfully.**

}

Rule 2: If a base class method declares to throw a checked exception, an overriding method in the derived class can choose not to declare to throw any checked exception

Examine the following code:

class Base {

public void aMethod() throws IOException {}

public void withRuntimeException() throws RuntimeException {}

}

class Derived1 extends Base {

public void aMethod() {}

public void withRuntimeException() {}//**Both lines compile successfully.**

}

Rule 3: If a base class method declares to throw a checked exception, an overriding method in the derived class cannot declare to throw a superclass of the exception thrown by the one in the base class

Examine the following code:

class Base {

public void aMethod() throws IOException {}

public void withRuntimeException() throws NullPointerException {}

}

class Derived2 extends Base {

public void aMethod() throws Exception {}//**fails to compile.**

public void withRuntimeException() throws RuntimeException{}

}

Tule 4: If a base class method declares to throw a checked exception, an overriding method in the derived class can declare to throw the same exception

The following code compiles successfully:

class Base {

void aMethod() throws IOException {}

void methodUncheckedEx() throws Error {}

}

class Derived3 extends Base {

void aMethod() throws IOException {}// **compile successfully.**

void methodUncheckedEx() throws NullPointerException {}

}

Rule 5: If a base class method declares to throw a checked exception, an overriding method in the derived class can declare to throw a derived class of the exception thrown by the one in the base class

The following code compiles successfully:

class Base {

void aMethod() throws IOException {}

}

class Derived4 extends Base {

void aMethod() throws FileNotFoundException {}// **compile successfully**

}

EXCEPTIONS THAT YOU CATCH IN A MULTI-CATCH BLOCK CAN’T SHARE AN INHERITANCE RELATIONSHIP

What happens if you add another line of code in the previous example, which involves reading from FileInputStream, which might throw an IOException? Let’s add IOException to the list of exceptions being caught in the multi-catch block:

class MultiCatch {

void myMethod(Connection con, String fileName) {

try {

File file = new File(fileName);// **May throw FileNotFoundException**

FileInputStream fin = new FileInputStream(file);

fin.read();//**May throw IOException**

Statement stmt = con.createStatement();//**May throw SQLException**

}

// **Fails to compile**

catch (**IOException| FileNotFoundException** | SQLException e) {

System.out.println(e.toString());

}

}

}

This code fails compilation with the following error message:

Alternatives in a multi-catch statement cannot be related by sub classing.

Looks like the code fails to compile because the IOException is caught before the FileNotFoundException. In regular catch blocks, if you catch a superclass exception before a derived class exception, the code won’t compile. So let’s swap the order of IOException and FileNotFoundException in the preceding code:

try {

//Exception code here

}

//**Swapping exception types doesn’t make a difference; code fails to compile.**

catch (**FileNotFoundException | IOException|** SQLException e) {

System.out.println(e.toString());

}

The correct multi-catch block for code that may throw an IOException, FileNotFoundException, and SQLException is as follows:

try {

//Exception code here

}

// **Catch IOException (superclass of FileNotFoundException) and SQLException or any of their subclasses.**

catch (**IOException | SQLException e**) {

System.out.println(e.toString());

}

}

Combining multi-catch and single-catch blocks

try {

//Exception code here.

}

//**Catch FileNotFoundException or any of its subclasses.**

catch (FileNotFoundException e) {}

//**Catch IOException and SQLException or any of their subclasses (except for FileNotFoundException or its subclasses).**

catch (IOException | SQLException e) {}

EXAM TIP Watch out for a combination of multi-catch and single-catch exception handlers on the exam. They can get quite tricky.

In multi-catch block we should use single variable.

catch (SQLException | IOException e ) {...}//**Compiles**

catch (SQLException el | IOException e2 ) {...}//**Fails to compile**

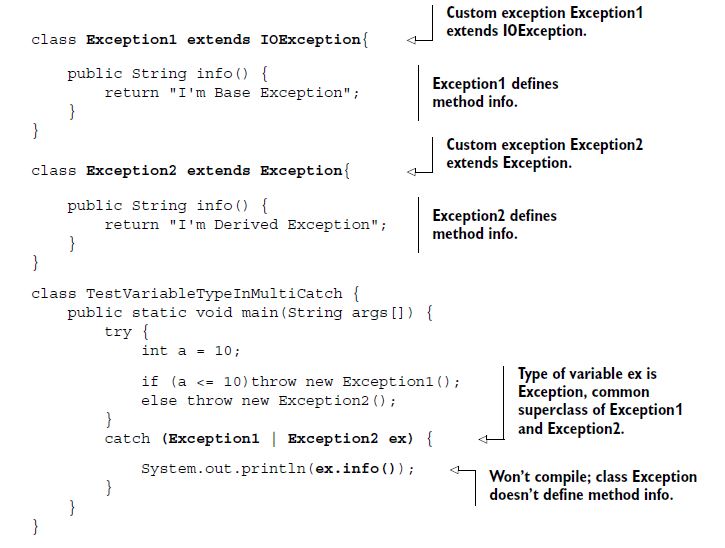
In a multi-catch block, variable e is implicitly final.

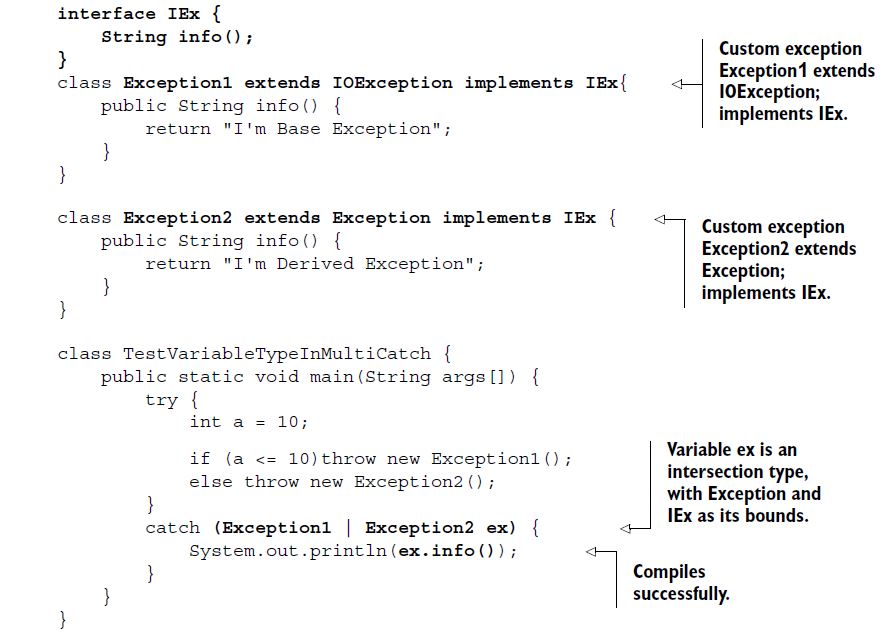
catch (IOException| SQLException e) {

**e = new FileNotFoundException();**//**Won’t compile**

}

Type of exception variable in a multi-catch block is Exception





You can call methods accessible to class Exception and interface IEx on the reference variable ex.

***Auto-closing resources with a try-with-resources***

try {

}

finally {

if (fis != null)

try {

fis.close();

}

catch (Exception e) {/\* \*/}

}

As you can see, closing a resource required a lot of boilerplate code and was error prone too. What if a developer didn’t close a resource in a finally block?

If you declare a resource by using a try-with-resources statement, it automatically *closes* the resource by calling its close method, *just* before the end of the try block. A resource must implement the java.lang.AutoCloseable interface or any of its sub interfaces to be eligible to be declared in a try-with-resources statement.

NOTE FileInputStream implements java.io.Closeable, which, starting with Java 7, extends java.lang.AutoCloseable. So FileInputStream is a valid resource to be used by the try statement.

**try (FileInputStream fin = new FileInputStream(file)){**

//.. some code

}

// **FileInputStream; can throw FileNotFoundException**

catch (FileNotFoundException e) {

System.out.println(e.toString());

}

But this code doesn’t compile and gives the following compilation error: AutoClose.java:7: error: unreported exception IOException; must be caught or declared to be thrown.

The try-with-resources statement calls method close() *just* before the completion of the try block. Method close() throws IOException we must handle it.

try (FileInputStream fin = new FileInputStream(file)){

//.. some code

}

catch (**IOException** e) {

System.out.println(e.toString());

}

EXAM TIP Watch out for questions on the exam that include or exclude a semicolon at the end of the resource defined by a try-with-resources statement. A try-with resources statement can declare multiple resources, which are separated by a semicolon. After the last resource declaration, a semicolon is optional.

***Suppressed exceptions***

In a try-with-resources statement, if both the code in the try block and close() throw an exception, the exception thrown by close() is *suppressed* by the exception thrown by the try block.

The resources initialized by the try-with-resources statement are automatically closed, just before the end of execution of the try block. This happens regardless of whether any exceptions are thrown or not thrown.

class RiverRaft implements AutoCloseable {

public RiverRaft() throws Exception {

System.out.println("Start raft");

}

public void crossRapid() throws Exception {

System.out.println("Cross rapid");

throw new Exception("Cross Rapid exception");

}

public void close() throws Exception {

System.out.println("Reach river bank");

}

}

The class SuppressedExceptions initializes an instance of RiverRaft by using a try-with-resources statement:

public class SuppressedExceptions {

public static void main(String[] args) throws Exception {

try ( RiverRaft raft = new RiverRaft(); ) {

raft.crossRapid();

}

catch (Exception e) {

System.out.println("Exception caught:" + e);

}

}

}

**Output:**

Start raft

Cross rapid

Reach river bank

Exception caught:java.lang.Exception: Cross Rapid exception

Now what happens if close() in RiverRaft also throws an exception? Which exception will be propagated to the exception handler? Will it be the exception from close() or from crossRapid()?

public class SuppressedExceptions {

public static void main(String[] args) throws Exception {

try ( RiverRaft raft = new RiverRaft(); ) {

raft.crossRapid();

}

catch (Exception e) {

System.out.println("Exception caught:" + e);

}

}

}

class RiverRaft implements AutoCloseable {

public RiverRaft() throws Exception {

System.out.println("Start raft");

}

public void crossRapid() throws Exception {

System.out.println("Cross rapid");

throw new Exception("Cross Rapid exception");

}

public void close() throws Exception {

System.out.println("Reach river bank");

**throw new Exception("Close exception");**

}

}

**Output:**

Start raft

Cross rapid

Reach river bank

Exception caught:java.lang.Exception: Cross Rapid exception

Because the output of the previous code snippets looks identical, what do you think happened to the exception thrown by method close()? This exception was *suppressed* by the exception thrown by crossRapid().

You can retrieve the suppressed exceptions by calling getSuppressed() on the exception that has suppressed the other exceptions. The getSuppressed() method returns an array containing all of the exceptions that were suppressed in order to deliver the exception thrown by crossRapid().

Throwable[] exs = e.getSuppressed();

if (exs.length>0)

System.out.println(exs[0]);//**Retrieves and prints first suppressed exception.**

The getSuppressed() never returns null. If there aren’t any suppressed expressions, the length of the returned array is 0.

EXAM TIP The variables defined in a try-with-resources statement are implicitly final.

The variables used to refer to resources are implicitly final variables. You must *declare* and *initialize* resources in the try-with-resources statement. You can’t define un-initialized resources:

try (FileInputStream fin; FileOutputStream fout;){//**Won’t compile;**

}

It’s acceptable to the Java compiler to initialize the resources in a try-with-resources statement to null, only as long as they aren’t being reassigned a value in the try block.

try (FileInputStream fin = null; FileOutputStream fout = null;){// **Compiles**

}

try (FileInputStream fin = null; FileOutputStream fout = null;){

fin = new FileInputStream(inFile);// **Won’t compile**

}

The resource declared by try-with-resources scope is limited to the try block.

try (FileInputStream fin = new FileInputStream(inFile);){

}

finally {

fin.read();//**Won’t compile out of scope.**

}

Exceptions that are thrown by method close() defined in the AutoCloseable and Closeable interfaces.

package java.lang;

public interface AutoCloseable {

void close() **throws Exception**;

}

package java.io;

public interface Closeable extends AutoCloseable {

public void close() **throws IOException**;

}

Method close() in the Closeable interface overrides method close() from the AutoCloseable base interface.

THE RESOURCES DECLARED WITH TRY-WITH-RESOURCES ARE CLOSED IN THE REVERSE ORDER OF THEIR DECLARATION

class MyResource implements AutoCloseable{

String name;

MyResource(String name) {

this.name = name;

System.out.println("Created:"+name);

}

public void close() {

System.out.println("Closed:"+name);

}

}

class TestAutoCloseOrder {

public static void main(String args[]) {

try (MyResource res1 = new MyResource("1");

MyResource res2 = new MyResource("2")){

System.out.println("within try-with-resources");

}

finally {

System.out.println("finally");

}

}

}

**Output:**

Created:1

Created:2

within try-with-resources

Closed:2

Closed:1

Finally