Table of Contents

Java Quick Review

This file file serves as your book's preface, a great place to describe your book's content and ideas.

Chapter I: Syntax

* Keywords

* Variables

* Operators

* String

1. Keywords

用于定义访问权限修饰符的关键字				
private	protected	public		
用于定义类，函数，变量修饰符的关键字				
abstract	final	static	synchronized	
用于定义类与类之间关系的关键字				
extends	implements			
用于定义建立实例及引用实例，判断实例的关键字				
new	this	super	instanceof	
用于异常处理的关键字				
try	catch	finally	throw	throws
用于包的关键字				
package	import			
其他修饰符关键字				
native	strictfp	transient	volatile	assert

2. Variables

2.1 Types: primitive type and reference type

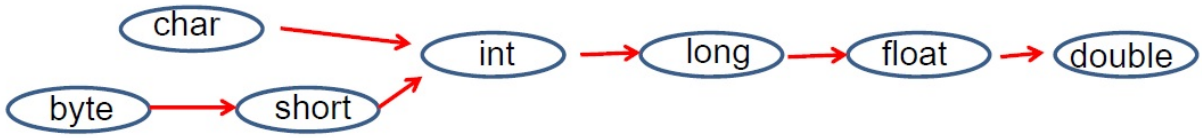
- primitive type: boolean, char, byte, short, int, long, float, double.
- reference type: class, interface, array [].

2.2 Primitive type

Type	Size	Range	Default Value
byte	1 byte	-128 to 127	0
char	2 byte	0 to 65536	'\u0000'
short	2 byte	-32768 to 32767	0
int	4 byte	-2^31 to 2^31-1 (about 2 billion)	0
long	8 byte	-2^63 to 2^63-1 (about 8*10^18)	0L
float	4 byte	3.4*10E38F	0.0F
double	8 byte	1.79*10E308	0.0
boolean	undefined	true or false	false

- Only instance variable has default value.
- For local variable, it must be initialized.
- All reference type (instance variable): default value is null.

2.2.1 Type Conversion



- byte, char, and short are all convert to int.
- boolean cannot be converted to any type.

Example: s += i vs s = s + i

- 1) `+=` compound operator will do implicity conversion, no compilation error. so when data is overflow, you will get a wrong result but code runs well.
- 2) `=` will do data type check, you will see compilation error if data types are mismatch in two sides

```
short s = 1;
int i = 123456;
s += i; // no compilation error, return wrong result (-7615)
s = s + i // compilation error
```

3. Operators

Operator Precedence

Operators	Precedence
postfix	*expr++ expr--*
unary	*++expr --expr +expr -expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Ex1: short-circuit

- The && and || operators "short-circuit", meaning they don't evaluate the right hand side if it isn't necessary.
- The & and | operators, when used as logical operators, always evaluate both sides.

```
public class Test {
    static int a = 5;
    static int b = 6;
    static int c = 7;
    static int d = 8;

    public static void main(String[] args) {
        testOR();
        testAND();
    }

    private static void testOR() {
        if (true | m1()) {
            System.out.println(a); // 10
        }
        // "||" will stop once it finds the first "true"
        if (true || m2()) {
            System.out.println(b); // 6
        }
    }
    private static void testAND() {
        if ((false & n1()) == false) {
            System.out.println(c); // 14
        }
        // "&&" will stop once it finds the first "false"
        if ((false && n2()) == false) {
            System.out.println(d); // 8
        }
    }
    private static boolean m1() {
        a = 10;
        return true;
    }
    private static boolean m2() {
        b = 12;
        return true;
    }
    private static boolean n1() {
        c = 14;
        return true;
    }
    private static boolean n2() {
        d = 16;
        return true;
    }
}
```

4. String

Ex1: Convert int to String

```
// bad
String s = intValue + "";

// good
String s = String.valueOf(intValue);
```

Ex2: Print string as chars

```
        String s = "Hello World";
        IntStream stream = s.chars();
        stream.forEach(p -> System.out.println(String.valueOf(p)));
```

Ex3: String concat

```
        String s1 = "Hello";
        String s2 = "World";

        //
        System.out.println(s1.concat(" ").concat(s2));

        // bad implementation
        String s = s1 + " " + s2;
        System.out.println(s);

       // Better solution
        StringBuilder sb = new StringBuilder();
        System.out.println(sb.append(s1).append(" ").append(s2).toString());

       // good solution (delimiter with " ")
        StringJoiner stringJoiner = new StringJoiner(" ");
        System.out.println(stringJoiner.add(s1).add(s2));

       // delimiter with ","
        StringJoiner sj = new StringJoiner(",");
        System.out.println(sj.add("one").add("two").add("three"));

       // delimiter with "," and start with "{" and end with "] "
        StringJoiner sj1 = new StringJoiner(",","{","]");
        System.out.println(sj1.add("one").add("two").add("three"));

        //
        System.out.println(String.join(",", "one", "two", "three"));
```

Ex4: **String equals**

```
//Bad:
args.equals("local");

//Good: avoid null pointer exception
"local".equals(args);

// google guava
Object.equals(obj1, obj2);
```

Ex5: **StringBuilder vs StringBuffer**

```
StringBuilder is faster but not thread-safe.
StringBuffer is slower but thread-safe.

Usually, StringBuilder is used to build string.
```

Chapter II: OO Concept

* 1. Class and Object

* 2. Encapsulation

* 3. Abstraction

* 4. Inheritance

* 5. Polymorphism

* 6. Interface

* 7. static, final, and nested classes

1. Class and Object

- Class is a blueprint which describes a category of objects.
- Object is instance which is concrete.

1.1 Access modifier

Four types: private / default / protected / public

- private: is only accessible inside the class.
- default: is accessible inside the package.
- protected: is accessible inside the package and sub-classes outside the package.
- public, is accessible everywhere.

1.2 Constructor

```
public People(){}
```

- It has the same name as Class.
- No return type (You cannot put any return type, even for void)
- If you don't explicitly define any constructor, system will provide a default one (empty args). Otherwise, system will not provide any constructor. Note, when spring IOC injects an object, it requires object has an explicit empty constructor.
- If you do not want the class to be constructed outside, use private `private People(){}`

2. Encapsulation (hiding data)

Encapsulation is one of the four fundamental OOP concepts:

```
* **Encapsulation**: hiding data. Often referred to as "Information Hiding", revolves
more specifically around internal data (fields / members representing the state) owned
 by a class instance, by enforcing access to the internal data in a controlled manner,
 and preventing direct, external change to these fields.
* **Abstraction**: occurs during class level design, with the objective of hiding the
implementation complexity of how the the features offered by an API / design / system
were implemented, in a sense simplifying the 'interface' to access the underlying impl
ementation.
* **Inheritance**: refers to using the structure and behavior of a superclass in a sub
class.
* **Polymorphism**: refers to changing the behavior of a superclass in the subclass.
```

Encapsulation is the technique of **making the fields in a class private** and **providing access to the fields via public methods**. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.

The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

```
public class EncapTest{
   private String name;
   private String idNum;
   private int age;

   public int getAge(){
      return age;
   }
   public String getName(){
      return name;
   }
   public String getIdNum(){
      return idNum;
   }

   //With setter(), you can put some defence conditions here. Otherwise, you lost the
control
   public void setAge( int newAge){
      if (newAge <= 0) {
        this.age = 0;
      } else {
        this.age = newAge;
      }
   }
   public void setName(String newName){
      this.name = newName;
   }
   public void setIdNum( String newId){
      this.idNum = newId;
   }
}
```

3. Abstraction

3.1 Abstract Class

```
* An abstract class is a class that is declared ``abstract`` —**it may or may not incl
ude abstract methods.**
* **Abstract classes cannot be instantiated, but they can be subclassed**.
```

3.2 Abstract Method

```
* An abstract method is a method that is declared without an implementation.
```java
 public boolean validateDate(String date);
 ```
* When an abstract class is subclassed, the subclass usually provides implementations
for all of the abstract methods in its parent class. However, if it does not, then the
 subclass must also be declared abstract.
```

Note: Methods in an interface (see the Interfaces section) that are not declared as default or static are implicitly abstract, so the abstract modifier is not used with interface methods. (It can be used, but it is unnecessary.)

3.3 static members

An abstract class may have static fields and static methods. You can use these static members with a class reference (for example, AbstractClass.staticMethod()) as you would with any other class.

3.4 Abstract Classes VS Interfaces

Abstract Classes	Interfaces
No restriction on fields	All fields are `public static final`
No restriction on methods	All methods are `public`
Only extends one class	Can **extend** multiple interfaces
Cannot be instantiated	Cannot be instantiated

```
* **Consider using abstract classes if any of these statements apply to your situation
**:
      * You want to share code among several closely related classes.
      * You expect that classes that extend your abstract class have many common metho
ds or fields, or  require access modifiers other than public (such as protected and pr
ivate).
      * You want to declare non-static or non-final fields. This enables you to define
 methods that can access and modify the state of the object to which they belong.

* **Consider using interfaces if any of these statements apply to your situation**:
      * You expect that unrelated classes would implement your interface. For example,
 the interfaces ``Comparable`` and ``Cloneable`` are implemented by many unrelated cla
sses.
      * You want to specify the behavior of a particular data type, but not concerned
about who implements its behavior.
      * You want to take advantage of multiple inheritance of type.
```

4. Inheritance

4.1 Subclass and Superclass

```
* A class that is derived from another class is called a **subclass** (also a derived
class, extended class, or child class).
* The class from which the subclass is derived is called a **superclass** (also a base
 class or a parent class).
* A subclass inherits all the members (fields, methods, and nested classes) from its s
uperclass. Constructors are not members, so they are not inherited by subclasses, but
the constructor of the superclass can be invoked from the subclass by ``super()``.
```

What you can do in a subclass

```
* The inherited fields can be used directly, just like any other fields.
* You can declare a field in the subclass with the same name as the one in the supercl
ass, thus **hiding it** (not recommended).
* You can declare new fields in the subclass that are not in the superclass.
* The inherited methods can be used directly as they are.
* You can write a new instance method in the subclass that has the same signature as t
he one in the superclass, thus **overriding it**. If you still want to use the method
implementation in superclass, you can use super to invoke it, like **super.methodA()**
.
* You can write a new static method in the subclass that has the same signature as the
 one in the superclass, thus **hiding it**.
* You can declare new methods in the subclass that are not in the superclass.
* You can write a subclass constructor that invokes the constructor of the superclass,
 either implicitly or by using the keyword **super**.
```

4.2 Private members in Superclass

```
* A subclass does not inherit the private members of its parent class. However, if the
 superclass has public or protected methods for accessing its private fields, these ca
n also be used by the subclass.

* A nested class has access to all the private members of its enclosing class—both fie
lds and methods. Therefore, a public or protected nested class inherited by a subclass
 has indirect access to all of the private members of the superclass.
```

4.3 Overriding and Hiding Methods

```
* An instance method in a subclass with the same signature and return type as an insta
nce method in the superclass overrides the superclass's method.

* If a subclass defines a static method with the same signature as a static method in
the superclass, then the method in the subclass hides the one in the superclass.

* **Rule 1: instance field and static member are invoked based on declared type.**

* **Rule 2: instance method is invoked based on assigned type.**

* **Rule 3: subclass cannot define a more restricted access modifier on an override me
thod than superclass's method.**
```

Example:

```
public class Animal {
    private int i = 10;
    public int j = 20;
    public static int z = 30;

    public static void testStatic() {
        System.out.println("calling super static method");
    }
    public void testInstanceMethod() {
        System.out.println("calling super instance method");
    }
    public void testAccess() {}
}

public class Cat extends Animal {

    public int j = 50;
    public static int z = 60;

    public static void testStatic() {
        System.out.println("calling sub static method");
    }
    public void testInstanceMethod() {
        System.out.println("calling sub instance method");
    }

    // Rule 3: error
    protected void testAccess(){}

    public static void main(String[] args) {
         Animal animal = new Animal();
        Cat cat = new Cat();
        Animal animalCat = cat;

        System.out.println(animalCat.j); // Rule 1:  call super.j 20
        System.out.println(animalCat.z); // Rule 1:  call super.z 30
        animalCat.testInstanceMethod(); // Rule 2: call sub.testInstanceMethod()
        animalCat.testStatic(); // Rule 1: call super.testStatic()
    }
}
```

4.4 Hiding fields

Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through **super**, which is covered in the next section. Generally speaking, we don't recommend hiding fields as it makes code difficult to read.

4.5 Keyword: super

```
* Accessing Superclass Members
```

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword super. You can also use super to refer to a hidden field (although hiding fields is discouraged).

```
* Subclasses Constructors
```

Example:

```
public MountainBike(int startHeight,
                    int startCadence,
                    int startSpeed,
                    int startGear) {
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
}
```

Rule: Invocation of a superclass constructor must be the first line in the subclass constructor.

4.6 Object as a Superclass

The Object class, in the `java.lang package`, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the Object class.

```
* protected Object clone() throws CloneNotSupportedException
  Creates and returns a copy of this object.
* public boolean equals(Object obj)
  Indicates whether some other object is "equal to" this one.
* protected void finalize() throws Throwable
  Called by the garbage collector on an object when garbage
  collection determines that there are no more references to the object
* public final Class getClass()
  Returns the runtime class of an object.
* public int hashCode()
  Returns a hash code value for the object.
* public String toString()
  Returns a string representation of the object.
```

The following methods all play a part in synchronizing the activities of independently running threads in a program:

```
* public final void notify()
* public final void notifyAll()
* public final void wait()
* public final void wait(long timeout)
* public final void wait(long timeout, int nanos)
```

4.6.1 hashCode() Method

The value returned by hashCode() is the object's hash code, which is the object's memory address in hexadecimal.

By definition, if two objects are equal, their hash code must also be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation of hashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well.

4.6.2 equals() Method

The equals() method compares two objects for equality and returns true if they are equal. For two objects, **if you do not override equals() and hashCode() , they will never be equal**. Because super equals() compares the hashCode which is the memory address of object.

4.6.3 finalize() Method

The Object class provides a callback method, finalize(), that may be invoked on an object when it becomes garbage. Object's implementation of finalize() does nothing—you can override finalize() to do cleanup, such as freeing resources.

The finalize() method may be called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect finalize() to close them for you, you may run out of file descriptors.

5. Polymorphism

5.1 Definition:

Polymprphism means many forms. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Example:

```
public abstract class Human{
   public abstract void goPee();
}
public class Male extends Human{
    @Override
    public void goPee(){
        System.out.println("Stand Up");
    }
}
public class Female extends Human{
    @Override
    public void goPee(){
        System.out.println("Sit Down");
    }
}
public static void main(String[] args){
    ArrayList<Human> group = new ArrayList<Human>();
    group.add(new Male());
    group.add(new Female());
    // ... add more...

    // tell the class to take a pee break
    for (Human person : group){  person.goPee(); }
}

//Running this would yield:
Stand Up
Sit Down
```

5.2 Compile time polymorphism (static binding or method overloading)

Method overloading, an object can have two or more methods with same name, BUT, with their method parameters different. If only return type is different for two methods, this is not allowed.

```
// Difference in parameter type
public static double Math.max(double a, double b){..}
public static float Math.max(float a, float b){..}
public static int Math.max(int a, int b){..}
public static long Math.max(long a, long b){..}

// Difference in parameter count
EmployeeFactory.create(String firstName, String lastName){...}
EmployeeFactory.create(Integer id, String firstName, String lastName){...}

//The following case is NOT allowed
public int test(int i) { return i;}
public void test(int i) { system.out.println(i)}
```

5.3 Runtime polymorphism (dynamic binding or method overriding)

Runtime polymorphism is essentially referred as method overriding. Method overriding is a feature which you get when you implement inheritance in your program.

*** Rule: Always invoked assigned object's instance method.**

6. Interface

6.1 Definition:

There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, interfaces are such contracts.

```
* Interface can contain only
      * fields are constants (public static final)
      * methods are public abstract, never final
      * **default methods(have method body)** // in Java 8
      * **static methods (have method body)** // in Java 8
      * nested types
      * Interfaces cannot be instantiated—they can only be **implemented** by classes
or **extended** by other interfaces.
```

6.2 Default Methods (Java 8)

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

6.3 Static Methods (Java 8)

You can define static methods in interfaces. (A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.) This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class.

6.4 Implementation

```
* Defining an interface
```

```
public interface TimeClient {
    //(public static final)
    double E = 2.718282;

    //abstract method: (public abstract)
    void doSomething (int i, double x);

    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                                int hour, int minute, int second);
    LocalDateTime getLocalDateTime();

    // static method (public): only in Java 8
    static ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    // default method (public): only in Java 8
    default ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }

}
```

```
* Implementing an interface
```

```
public class TestSimpleTimeClient {
    public static void main(String[] args) {
        TimeClient myTimeClient = new SimpleTimeClient();
        System.out.println("Current time: " + myTimeClient.toString());
        System.out.println("Time in California: " +
            myTimeClient.getZonedDateTime("Blah blah").toString());
    }
}
```

```
* Using an Interface as a Type
```

If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.

7. Static, Final, and Nested Classes

7.1 Static

7.1.1 Static Method

```
* You can only access class's static fields or methods, non-static fields and methods
are in-accessible.
* **this / super** cannot be used in static method,because you don't need an instance
to access static method.
```

7.1.2 Static Block When a class is loaded, its static block will be executed **(only once)** before main method. Static block is usually used for class's initialization.

Example 1: Singleton

```
class Source {
    private static Connection conn = new Connection();

    private Source() {}

    public static Connection getConnection() {
        return conn;
    }
}
```

Example 2: main method

```
// 1) JVM needs to invoke main mehtod, so the access modifier must be public.
// 2) JVM does not create an instace for the class, so it must be a static method.
// 3) argument is a String array.
public static void main(String[] args) {
}
```

7.2 Final

`final` can be used on variable, method, and class

```
* On variable: **final fields or local variables** are constants, can only be assigned
 once. And final variables must have assigned value when they are declared or be assig
ned in constructor explicitly.
* On method: indicate that the method **cannot be overridden** by subclasses. The Obje
ct class does this—a number of its methods are final.
* On class: indicate that the class **cannot be subclassed**. This is particularly use
ful, for example, when creating an immutable class like the String class. Example:  Th
e wrapper classes for the primitive types:``java.lang.Integer ,java.lang.Byte ,java.la
ng.Character ,java.lang.Short ,java.lang.Boolean ,java.lang.Long ,java.lang.Double ,ja
va.lang.Float ``
```

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make the getFirstPlayer method in this ChessAlgorithm class final:

```
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    final ChessPlayer getFirstPlayer() { return ChessPlayer.WHITE; }
}
```

Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

7.3 Nested Classes (one implementation of encapsulation)

7.3.1 Definition: A nested class is a class which is defined within another class.

```
* Nestes classes
    * static nested classes
    * non-static nested classes (inner classes)
        * local classes
        * anonymous classes
* Non-static nested classes (inner classes) have access to other members of the enclos
ing class, even if they are declared private.
    * **Its name must be different from the class's name.**
    * **Its members cannot be declared as static.**

* Static nested classes do not have access to other members of the enclosing class.
    * **It can use them only through an object reference.**
```

As a member of the OuterClass, a nested class can be declared private, public, protected, or package private. (Recall that outer classes can only be declared public or package private)

```
class OuterClass {
    static class StaticNestedClass {    }
    class InnerClass { }
}
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

There are two special kinds of inner classes: local classes and anonymous classes.

```
* Local classes: declare an inner class within the body of a method.
* Annonymous classes: declare an inner class within the body of a method without namin
g the class.
```

7.3.2 When to Use Nested Classes, Local Classes, Anonymous Classes, and Lambda Expressions

```
* Nested Classes enable you to logically group classes that are only used in one place
, increase the use of encapsulation, and create more readable and maintainable code.

* Local classes, anonymous classes, and lambda expressions also impart these advantage
s; however, they are intended to be used for more specific situations:

* Local class: Use it if you need to create more than one instance of a class, access
its constructor, or introduce a new, named type (because, for example, you need to inv
oke additional methods later).

* Anonymous class: Use it if you need to declare fields or additional methods.

* Lambda expression: Use it if you are encapsulating a single unit of behavior that yo
u want to pass to other code. For example, you would use a lambda expression if you wa
nt a certain action performed on each element of a collection, when a process is compl
eted, or when a process encounters an error. Use it if you need a simple instance of a
 functional interface and none of the preceding criteria apply (for example, you do no
t need a constructor, a named type, fields, or additional methods).

* Nested class: Use it if your requirements are similar to those of a local class, you
 want to make the  type more widely available, and you don't require access to local v
ariables or method parameters.

* Use a non-static nested class (or inner class) if you require access to an enclosing
 instance's all fields and methods.

* Use a static nested class if you don't require this access.
```

Chapter III: Exception

1.1 Exception

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

- **Checked Exception(compile time exception)**: Applications are expected to be able to catch and meaningfully do something with the rest, such as `FileNotFoundException` and `TimeoutException`. Compiler forces client handle this exception (either catch the exception or declare it in a throws clause.)
- **Unchecked Exception**:
  - **RuntimeException**: Runtime exceptions are those exceptions that are thrown at runtime because of either wrong input data or because of wrong business logic etc. These are not checked by the compiler at compile time.
  - **Error**: It is mostly thrown by JVM which is fatal and **there is no way for the application to recover from that error**. For instance: `OutOfMemoryError`. **Errors should not be caught or handled.**

1.2 Exception Examples

- `IOException` (checked)
  - `FileNotFoundException`
  - `EOFException`
  - `FileSystemException`
- `RuntimeException` (unchecked)
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `ArithmeticException`

1.3 Handle Exception

Two ways:

- try / catch / finally
- throws

1.3.1 try / catch / finally

- You may have one try, one finally, and multiple catches. Note: exception2 must be exception1's superclass.
- finally block always be executed even try have a return clause.
- For an overridden method in subclass, its defined catch exception must be same or a derived exception to the exception in superclass.
- If the exception from try block is caught by catch block, it looks like catch block eats the exception and continue to run the rest of code. If not, the app will be interrupted and stopped. The rest of code won't be executed.
- The finally block may not execute if
  - System.exit() is called. So the JVM exits while the try or catch code is being executed.
  - JVM crashes. Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.
  - The try{} block never ends.

```
try {}
catch(e1){}
catch(e2){}
 .
 .
catch(en){}
finally{}
```

1.3.2 Throws

```
public class A {
    public void methodA() throws IOException {}
}

public class B1 extends A {
    public void methodA() throws FileNotFoundException {} // this exception has to be
IOException or its subclasses
}

public class B2 extends A {
    public void methodA() throws Exception { //error, because Exception is the supercl
ass of IOException}
}
```

1.4 Customize Exception

```
class MyException extends Exception {
    private int idnumber;
    public MyException(String message, int id) {
        super(message);
        this.idnumber = id;
    }
    public int getId() {
        return idnumber;
    }
}


public class Test{
    public void regist(int num) throws     MyException {
        if (num < 0) {
            throw new MyException(“num cannot be negative”,3);
        }
        else {
            System.out.println("Regitster: " + num );
        }
    }
    public void manager() {
        try {
            regist(100);
        } catch (MyException e) {
            System.out.print("Register error: "+e.getId());
        }
            System.out.print("Register ended");
    }
    public static void main(String args[]){
            Test test = new Test();
            t.manager();
    }
}
```

2. The try-with-resources statement

The try-with-resources statement is a try statement that declares **one or more resources**. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable`, can be used as a resource.

2.1 Example

Reads the first line from a file. It uses an instance of BufferedReader to read data from the file. BufferedReader is a resource that must be closed after the program is finished with it:

```
// example 1:
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}

// example 2:
public static void viewTable(Connection con) throws SQLException {
    String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";

    try (Statement stmt = con.createStatement()) {
        ResultSet rs = stmt.executeQuery(query);

        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
        }
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    }
}
```

The class BufferedReader, in Java SE 7 and later, implements the interface `java.lang.AutoCloseable`. Because the BufferedReader instance is declared in a try-with-resource statement, it will be closed regardless of whether the try statement completes normally or abruptly.

2.2 try-with-resources VS finally{}

You can use a finally block to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly.

```
static String readFirstLineFromFileWithFinallyBlock(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine(); // suppressed
    } finally {
        if (br != null) br.close(); // throws exception
    }
}
```

In this example, if the methods `readLine()` and `close()` both throw exceptions, then the method `readFirstLineFromFileWithFinallyBlock()` **throws the exception thrown from the finally block; the exception thrown from the try block is suppressed**.

In contrast, in the example `readFirstLineFromFile()`, if exceptions are thrown from both the try block and the try-with-resources statement, then the method readFirstLineFromFile **throws the exception thrown from the try block; the exception thrown from the try-with-resources block is suppressed**.

2.3 Multi try-with-resources

You may declare one or more resources in a try-with-resources statement.

```
public static void writeToFileZipFileContents(String zipFileName,String outputFileName)
 throws java.io.IOException {

    java.nio.charset.Charset charset = java.nio.charset.StandardCharsets.US_ASCII;
    java.nio.file.Path outputFilePath = java.nio.file.Paths.get(outputFileName);

    // Open zip file and create output file with
    // try-with-resources statement

    try (
        java.util.zip.ZipFile zf = new java.util.zip.ZipFile(zipFileName);
        java.io.BufferedWriter writer = java.nio.file.Files.newBufferedWriter(outputFi
lePath, charset)
    ) {
        // Enumerate each entry
        for (java.util.Enumeration entries = zf.entries(); entries.hasMoreElements();)
 {
            // Get the entry name and write it to the output file
            String newLine = System.getProperty("line.separator");
            String zipEntryName = ((java.util.zip.ZipEntry)entries.nextElement()).getN
ame() + newLine;
            writer.write(zipEntryName, 0, zipEntryName.length());
        }
    }
}
```

When the block of code that directly follows it terminates, either normally or because of an exception, the close methods of the BufferedWriter and ZipFile objects are automatically called in this order. Note that the close methods of resources are called in the opposite order of their creation.

Note: A try-with-resources statement can have catch and finally blocks just like an ordinary try statement. **In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed**.

2.4 Suppressed Exceptions

An exception can be thrown from the block of code associated with the try-with-resources statement. In the example writeToFileZipFileContents, an exception can be thrown from the try block, and up to two exceptions can be thrown from the try-with-resources statement when it tries to close the ZipFile and BufferedWriter objects. If an exception is thrown from the try block and one or more exceptions are thrown from the try-with-resources statement, then those exceptions thrown from the try-with-resources statement are suppressed, and the exception thrown by the block is the one that is thrown by the writeToFileZipFileContents method. You can retrieve these suppressed exceptions by calling the Throwable.getSuppressed method from the exception thrown by the try block.

2.5 Classes That Implement the AutoCloseable or Closeable Interface

See the Javadoc of the AutoCloseable and Closeable interfaces for a list of classes that implement either of these interfaces. The Closeable interface extends the AutoCloseable interface. The close method of the Closeable interface throws exceptions of type IOException while the close method of the AutoCloseable interface throws exceptions of type Exception. Consequently, subclasses of the AutoCloseable interface can override this behavior of the close method to throw specialized exceptions, such as IOException, or no exception at all.

3 Advantages of Exceptions

3.1 Advantage 1: Separating Error-Handling Code from "Regular" Code

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

At first glance, this function seems simple enough, but it ignores all the following potential errors.

```
* What happens if the file can't be opened?
* What happens if the length of the file can't be determined?
* What happens if enough memory can't be allocated?
* What happens if the read fails?
* What happens if the file can't be closed?
```

To handle such cases, the `readFile()` function must have more code to do error detection, reporting, and handling. Here is an example of what the function might look like.

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere. If the `readFile()` function used exceptions instead of traditional error-management techniques, it would look more like the following.

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.

3.2 Advantage 2: Propagating Errors Up the Call Stack

A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. Suppose that the `readFile()` method is the fourth method in a series of nested method calls made by the main program: method1 calls method2, which calls method3, which finally calls readFile.

```
method1 {
    call method2;
}

method2 {
    call method3;
}

method3 {
    call readFile;
}
```

Suppose also that method1 is the only method interested in the errors that might occur within readFile. Traditional error-notification techniques force method2 and method3 to propagate the error codes returned by readFile up the call stack until the error codes finally reach method1—the only method that is interested in them.

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

Recall that the Java runtime environment searches backward through the call stack to find any methods that are interested in handling a particular exception. A method can duck any exceptions thrown within it, thereby allowing a method farther up the call stack to catch it. Hence, only the methods that care about errors have to worry about detecting errors.

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

However, as the pseudocode shows, ducking an exception requires some effort on the part of the middleman methods. Any checked exceptions that can be thrown within a method must be specified in its throws clause.

3.3 Advantage 3: Grouping and Differentiating Error Types

Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy. An example of a group of related exception classes in the Java platform are those defined in java.io — IOException and its descendants. IOException is the most general and represents any type of error that can occur when performing I/O. Its descendants represent more specific errors. For example, FileNotFoundException means that a file could not be located on disk.

A method can write specific handlers that can handle a very specific exception. The FileNotFoundException class has no descendants, so the following handler can handle only one type of exception.

```
catch (FileNotFoundException e) {
    ...
}
```

A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement. For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an IOException argument.

```
catch (IOException e) {
    ...
}
```

This handler will be able to catch all I/O exceptions, including FileNotFoundException, EOFException, and so on. You can find details about what occurred by querying the argument passed to the exception handler. For example, use the following to print the stack trace.

```
catch (IOException e) {
    // Output goes to System.err.
    e.printStackTrace();
    // Send trace to stdout.
    e.printStackTrace(System.out);
}
```

You could even set up an exception handler that handles any Exception with the handler here.

```
// A (too) general exception handler
catch (Exception e) {
    ...
}
```

The Exception class is close to the top of the Throwable class hierarchy. Therefore, this handler will catch many other exceptions in addition to those that the handler is intended to catch. You may want to handle exceptions this way if all you want your program to do, for example, is print out an error message for the user and then exit.

In most situations, however, you want exception handlers to be as specific as possible. The reason is that the first thing a handler must do is determine what type of exception occurred before it can decide on the best recovery strategy. In effect, by not catching specific errors, the handler must accommodate any possibility. Exception handlers that are too general can make code more error-prone by catching and handling exceptions that weren't anticipated by the programmer and for which the handler was not intended.

As noted, you can create groups of exceptions and handle exceptions in a general fashion, or you can use the specific exception type to differentiate exceptions and handle exceptions in an exact fashion.

4. Best Practice for Exception

4.1 implement a re-try-catch

Ex1:

```
int count = 0;
int maxTries = 3;
while(true) {
    try {
        // Some Code
        // break out of loop, or return, on success
    } catch (SomeException e) {
        // handle exception
        if (++count == maxTries) { throw e;}
    }
}
```

Ex2:

```
//You can use AOP and Java annotations from jcabi-aspects
@RetryOnFailure(attempts = 3, delay = 5)
public String load(URL url) {
  return url.openConnection().getContent();
}
```

4.2 Always add log in catch block

Example one:

```
public void consumeAndForgetAllExceptions(){
    try {
        ...some code that throws exceptions
    } catch (Exception ex){
        ex.printStacktrace();
    }
}
```

What is wrong with the code above? Once an exception is thrown, normal program execution is suspended and control is transferred to the catch block. The catch block catches the exception and just suppresses it. Execution of the program continues after the catch block, as if nothing had happened.

Example two:

```
public void someMethod() throws Exception{
}
```

This method is a blank one; it does not have any code in it. How can a blank method throw exceptions? Java does not stop you from doing this. Recently, I came across similar code where the method was declared to throw exceptions, but there was no code that actually generated that exception. When I asked the programmer, he replied "I know, it is corrupting the API, but I am used to doing it and it works."

Broadly speaking, there are three different situations that cause exceptions to be thrown:

- Exceptions due to programming errors(**Runtime or unchecked exception**): In this category, exceptions are generated due to programming errors (e.g., `NullPointerException` and `IllegalArgumentException`). The client code usually cannot do anything about programming errors.
- Exceptions due to client code errors (**checked exception**): Client code attempts something not allowed by the API, and thereby violates its contract. The client can take some alternative course of action, if there is useful information provided in the

exception. For example: an exception is thrown while parsing an XML document that is not well-formed. The exception contains useful information about the location in the XML document that causes the problem. The client can use this information to take recovery steps.

- Exceptions due to resource failures: Exceptions that get generated when resources fail. For example: the system runs out of memory or a network connection fails. The client's response to resource failures is context-driven. The client can retry the operation after some time or just log the resource failure and bring the application to a halt. Types of Exceptions in Java

Java defines two kinds of exceptions:

- Checked exceptions: Exceptions that inherit from the Exception class are checked exceptions. It occurs at compile time. Client code has to handle the checked exceptions thrown by the API, either in a catch clause or by forwarding it outward with the throws clause.
- Unchecked exceptions: RuntimeException also extends from Exception. However, all of the exceptions that inherit from RuntimeException get special treatment. There is no requirement for the client code to deal with them, and hence they are called unchecked exceptions.
- The difference between throws and throw keywords, throws is used to postpone the handling of a checked exception and throw is used to invoke an exception explicitly.

A checked exception thrown by a lower layer is a forced contract on the invoking layer to catch or throw it. The checked exception contract between the API and its client soon changes into an unwanted burden if the client code is unable to deal with the exception effectively. Programmers of the client code may start taking shortcuts by suppressing the exception in an empty catch block or just throwing it and, in effect, placing the burden on the client's invoker. Checked exceptions are also accused of breaking encapsulation. Consider the following:

```
public List getAllAccounts() throws
    FileNotFoundException, SQLException{
    ...
}
```

The method getAllAccounts() throws two checked exceptions. The client of this method has to explicitly deal with the implementation-specific exceptions, even if it has no idea what file or database call has failed within getAllAccounts(), or has no business providing filesystem or database logic. Thus, the exception handling forces an inappropriately tight coupling between the method and its callers.

Chapter IV: Collections

1. Interface Structure

2. The Collection Interface

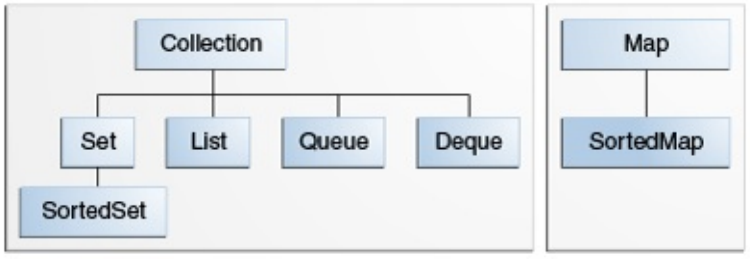
3. Set

4. List

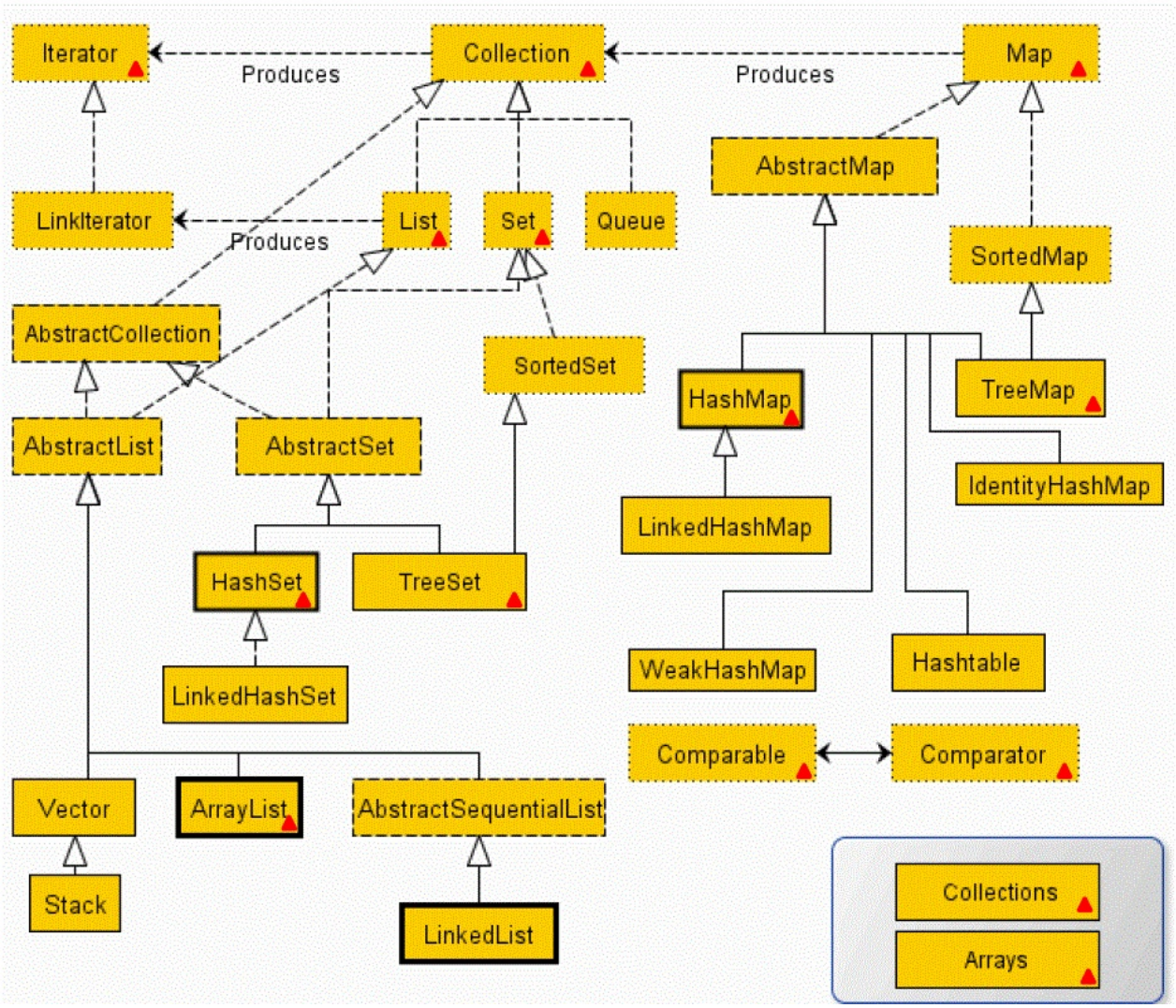
5. Map

1. Interface Structure

Core Interfaces:



Completed Inheritance Structure:



1.1 Collection

The root of the collection hierarchy. A collection represents a group of objects known as its elements. Collections may have one or some following properties:

- Allow duplicate elements
- Do not allow duplicate elements: **Set** / **Map** (no duplicate keys)
- Maintain elements inserted order
- Do not maintain elements inserted order
- Sorted
- Unsorted
- First-in, first-out (FIFO)
- Last-in, first-out (LIFO)

1.2 Set

A collection that cannot contain duplicate elements.

1.3 List

An ordered collection (sometimes called a sequence). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).

1.4 Queue

A collection used to hold multiple elements prior to processing. It can order elements in a FIFO (first-in, first-out) manner.

1.5 Deque

A collection used to hold multiple elements prior to processing. Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). It implements both stacks and queues at the same time.

1.6 Map

An object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value.

The last two core collection interfaces are merely sorted versions of Set and Map:

1.7 SortedSet

A Set that maintains its elements in ascending order.

1.8 SortedMap

A map that maintains its mappings in ascending key order. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

2. The Collection Interface

2.1 Features

- Store a group of objects.

- Varied-size
- **Cannot store primitive type data**
- **Un-synchronized** : basically all collection type are un-synchronized. Only these three are synchronized: Hashtable (not from collection), Vector, Stack.
- To make a collection synchronized: `Collections.synchronizedCollection(Collection<T> c)`

2.2 Traversing Collections

- Use for-each construct
- Use Iterators
- Use aggregate operations (JDK 8)

2.2.1 for-each construct

```
for (Object o : collection) { system.out.println(o)}
```

2.2.2 Iterators

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

The remove method removes the last element that was returned by next. The remove method may be called only once per call to next and throws an exception if this rule is violated. **Note that Iterator.remove is the only safe way to modify a collection during iteration.**

Use Iterator instead of the for-each construct when you need to:

- Remove the current element. The for-each construct hides the iterator, so you cannot call remove. **Therefore, the for-each construct is not usable for filtering.**
- Iterate over multiple collections in parallel.

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

2.2.3 Aggregate Operations (Java 8)

```
myShapesCollection.stream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e.getName()));
```

2.3 APIs

- Bulk Operations
- Array Operations

2.3.1 Bulk Operations

- **containsAll**: returns true if the target Collection contains all of the elements in the specified Collection.
- **addAll**: adds all of the elements in the specified Collection to the target Collection.
- **removeAll**: removes from the target Collection all of its elements that are also contained in the specified Collection.
- **retainAll**: removes from the target Collection all its elements that are not also contained in the specified Collection. That is, it retains only those elements in the target Collection that are also contained in the specified Collection.
- **clear**: removes all elements from the Collection.

2.3.2

- toArray()
- Arrays.asList()

The toArray methods are provided as a bridge between collections and older APIs that expect arrays on input. The array operations allow the contents of a Collection to be translated into an array.

```
// c is a Collection. Note:
Object[] a = c.toArray();

String[] a = c.toArray(new String[0]);
```

Arrays.asList(): Returns a fixed-size list backed by the specified array. (Changes to the returned list "write through" to the array.) This method acts as bridge between array-based and collection-based APIs, in combination with Collection.toArray(). The returned list is serializable and implements RandomAccess.

```
    List<String> stooges = Arrays.asList("Larry", "Moe", "Curly");

    stooges.add("Don"); // error(UnsupportedOperationException), stooges is  an array
object
```

Parameters:

- a - the array by which the list will be backed
- Returns:**a list view of the specified array**. It is not a list. So you cannot implement stooges.add().

2.4 How to write HashCode

```
hashCode = 31 * hashCode + (father == null ? 0 : father.hashCode());
hashCode = 31 * hashCode + depth;
hashCode = 31 * hashCode + cost;
hashCode = 31 * hashCode + matrix.hashCode();
hashCode = 31 * hashCode + java.util.Arrays.hashCode(coordinates);
```

3. Set

Set does not contain duplicate elements

- Un-sorted: HashSet (do not keep insert order)
- Un-sorted: LinkedHashSet (keep insert order)
- Sorted: TreeSet

3.1 HashSet

- Use hashCode and equals to check uniqueness (element must override hashCode and equals methods)
  - If hashCode is different, save element into HashSet. No need to check equals().
  - If hashCode is same, but not equals, save element into HashSet.
  - If both hashCode and equals are same, duplicate elements. Do not save it.
  - HashSet permits null.

3.2 TreeSet

- Sorting method 1: Element itself comparable
  - Element implements Comparable interface

    - Override compareTo()
    - If element itself has nature order, like String, no need to implement Comparable interface.
- Sorting method 2: Construct TreeSet with Comparator
    - Create a customized comparator class which implements Comparator interface
    - Override compare() method
    - Note: TreeSet's comparator will override object's compareTo() method.

```
public class ComparatorByLength implements Comparator {
    @Override
    public int compare(Object o1, Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;

        int temp = s1.length()-s2.length();
        return temp==0? s1.compareTo(s2): temp;
        // return 1; // keep inserting order
        // return -1 // keep inverse-inserting order
    }
}

TreeSet ts = new TreeSet(new ComparatorByLength());
```

4. List

- Vector:synchronized, CRUD performance is low.
- ArrayList:un-synchronized, CRUD performance is faster than Vector (designed to replace vector). Read is faster than LinkedList.
- LinkedList:un-synchronized, add and delete are faster than ArrayList ◦
- LinkedList VS ArrayList
    - LinkedList store elements within a doubly-linked list data structure.
    - LinkedList allows for constant-time insertions or removals, but only sequential access of elements.
    - LinkedList costs more memory
    - ArrayList: allow random access, so you can grab any element in constant time.
    - ArrayList: adding and removing are slower than LinkedList.

5. Map

- Map：adds (key, value), whereas Collection adds element.
- **key must be unique**

5.1 APIs

- Add

```
put(K key, V value)
// Return: Associates the specified value with the specified key in this map (opti
onal // operation).
```

- Delete

```
  void  clear()// Removes all of the mappings from this map (optional operation).
  value remove(key)//Removes the mapping for a key from this map if it is present
(optional operation).
```

- Check

```
  boolean containsKey(key)//Returns true if this map contains a mapping for the sp
ecified key.
  boolean containsValue(value)//Returns true if this map maps one or more keys to
the specified value.
  boolean isEmpty()//Returns true if this map contains no key-value mappings.
```

- Get

```
  value get(key)//Returns the value to which the specified key is mapped, or null
if this map contains no mapping for the key.
  int size()//Returns the number of key-value mappings in this map.
```

5.2 Implementations

- Hashtable :Synchronized, null cannot be key or value.
- HashMap : Un-synchronized and permits null。
- LinkedHashMap: it is similar to HashMap, but keeps insert order.
- TreeMap : The TreeMap is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.
- HashMap and HashTable check key (check both hasCode and equals)

```
if (key1.hashCode() == key2.hashCode() && key1.equals(key2)) { return true;}
```

- HashMap and HashTable check value (only check equals)

```
if (value1.equals(value2)) { return true;}
```

6. Queue

Queue Interface Structure

Type of Operation	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Queue is empty	Queue is empty
add(e): throws IllegalStateException	offer(e): return false
remove(): throws NoSuchElementException	poll(): return null
element(): throws NoSuchElementException	peek(): return null

7. Deque

Deque Methods

Type of Operation	First Element (Beginning of the `Deque` instance)	Last Element (End of the `Deque` instance)
Insert	`addFirst(e)` `offerFirst(e)`	`addLast(e)` `offerLast(e)`
Remove	`removeFirst()` `pollFirst()`	`removeLast()` `pollLast()`
Examine	`getFirst()` `peekFirst()`	`getLast()` `peekLast()`

8. Synchronization, Fail-fast and Fail-safe Iterators

8.1 Write a synchronized list

```
List list = new ArrayList();//un-synchronized
list = MyCollections.synList(list);// return a synchronized list.
// add synchronization
class MyCollections{
    public static  List synList(List list){
        return new MyList(list);
    }
    private class MyList implements List{
    private List list;
    private static final Object lock = new Object();
    MyList(List list){
        this.list = list;
    }

    public boolean add(Object obj){
        synchronized(lock){ return list.add(obj);}
    }

    public boolean remove(Object obj){
        synchronized(lock){ return list.remove(obj);}
    }
}
}
```

8.2 Fail-fast and Fail-safe Iterators

Concurrent Modification: When more threads are iterating over a collection, in between, one thread changes the structure of the collection (either adding the element to the collection or by deleting the element in the collection or by updating the value at particular position in the collection) is known as Concurrent Modification.

NOTE: structural modification is any operation that **ADD** or **delete** element;
NOT A STRUCTURAL MODIFICATION: merely setting the value of an element (in case of list) or changing the value associated with an existing key (in case of map).

- Fail-fast iterator: when add or delete concurrent modification applies on a un-synchronized collection, fail-fast iterator throws ConcurrentModificationException.
- Fail-safe iterator: allows add or delete concurrent modification applies on a concurrent version of collection. Fail Safe Iterator makes copy of the internal data structure (object array) and iterates over the copied data structure. So , original data structure remains structurally unchanged. Hence , no ConcurrentModificationException throws by the fail safe iterator.

```
public class FailFastExample{
    public static void main(String[] args){
        Map<String,String> premiumPhone = new HashMap<String,String>();
        premiumPhone.put("Apple", "iPhone");
        premiumPhone.put("HTC", "HTC one");
        premiumPhone.put("Samsung","S5");

        Iterator iterator = premiumPhone.keySet().iterator();

        while (iterator.hasNext()){
            System.out.println(premiumPhone.get(iterator.next()));
            premiumPhone.put("Sony", "Xperia Z");
        }
    }
}

// Output:
iPhone
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
        at FailFastExample.main(FailFastExample.java:20)
```

```
public class FailSafeExample{
    public static void main(String[] args){
        ConcurrentHashMap<String,String> premiumPhone =
                                new ConcurrentHashMap<String,String>();
        premiumPhone.put("Apple", "iPhone");
        premiumPhone.put("HTC", "HTC one");
        premiumPhone.put("Samsung","S5");

        Iterator iterator = premiumPhone.keySet().iterator();

        while (iterator.hasNext()){
            System.out.println(premiumPhone.get(iterator.next()));
            premiumPhone.put("Sony", "Xperia Z");
        }
    }
}
// Output:
S5
HTC one
iPhone
```

	Fail Fast Iterator	Fail Safe Iterator
Throw ConcurrentModification Exception	Yes	No
Clone object	No	Yes
Memory Overhead	No	Yes
Examples	HashMap,Vector,ArrayList,HashSet	CopyOnWriteArrayList, ConcurrentHashMap

9. Collections

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

It provides manipulations on Set, List, Map, etc.

9.1 Sort Operation

- reverse(List)
- shuffle(List)
- sort(List)
- sort(List，Comparator)
- swap(List，int， int)

9.2 Search and Replace

- Object max(Collection)
- Object max(Collection，Comparator)
- Object min(Collection)
- Object min(Collection，Comparator)
- int frequency(Collection，Object)
- boolean replaceAll(List list，Object oldVal，Object newVal)

9.3 Synchronized Methods

```
Collections.synchronizedCollection(Collection<T> c)
Collections.synchronizedList(List<T> c)
Collections.synchronizedSet(Set<T> c)
Collections.synchronizedSortedSet(SortedSet<T> c)
Collections.synchronizedMap(Map<T> c)
Collections.synchronizedSortedMap(SortedMap<T> c)
```

9.4 Algorithms

The polymorphic algorithms described here are pieces of reusable functionality provided by the Java platform. All of them come from the Collections class, and all take the form of static methods whose first argument is the collection on which the operation is to be performed. The great majority of the algorithms provided by the Java platform operate on List instances, but a few of them operate on arbitrary Collection instances. This section briefly describes the following algorithms:

- Sorting
- Shuffling
- Routine Data Manipulation
- Searching
- Composition
- Finding Extreme Values

Sorting
The sort algorithm reorders a List so that its elements are in ascending order according to an ordering relationship.

- sort(List)
- sort(List，Comparator)

The sort operation uses a slightly optimized **merge sort** algorithm that is fast and stable:

- **Fast**: It is guaranteed to run in n log(n) time and runs substantially faster on nearly sorted lists. Empirical tests showed it to be as fast as a highly optimized quicksort. A quicksort is generally considered to be faster than a merge sort but isn't stable and doesn't guarantee n log(n) performance.
- **Stable**: It doesn't reorder equal elements. This is important if you sort the same list repeatedly on different attributes. If a user of a mail program sorts the inbox by mailing date and then sorts it by sender, the user naturally expects that the now-contiguous list of messages from a given sender will (still) be sorted by mailing date. This is guaranteed only if the second sort was stable.
- **Quicksort** can be used to primitive type data which does not have stable issue.

The following trivial program prints out its arguments in lexicographic (alphabetical) order.

```
public class Sort {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

Let's run the program.

```
% java Sort i walk the line
```

The following output is produced.

```
[i, line, the, walk]
```

The second form of sort takes a `Comparator` in addition to a List and sorts the elements with the Comparator.

Recall that the anagram groups are stored as values in a Map, in the form of List instances. The revised printing code iterates through the Map's values view, putting every List that passes the minimum-size test into a List of Lists. Then the code sorts this List, using a Comparator that expects List instances, and implements reverse size-ordering. Finally, the code iterates through the sorted List, printing its elements (the anagram groups). The following code replaces the printing code at the end of the main method in the Anagrams example.

```
// Make a List of all anagram groups above size threshold.
List<List<String>> winners = new ArrayList<List<String>>();
for (List<String> l : m.values())
    if (l.size() >= minGroupSize)
        winners.add(l);

// Sort anagram groups according to size
Collections.sort(winners, new Comparator<List<String>>() {
    public int compare(List<String> o1, List<String> o2) {
        return o2.size() - o1.size();
    }});

// Print anagram groups.
for (List<String> l : winners)
    System.out.println(l.size() + ": " + l);
```

Running the program on the same dictionary as in The Map Interface section, with the same minimum anagram group size (eight), produces the following output.

```
12: [apers, apres, asper, pares, parse, pears, prase, presa, rapes, reaps, spare, spea
r]
11: [alerts, alters, artels, estral, laster, ratels,salter, slater, staler, stelar, ta
lers]
10: [least, setal, slate, stale, steal, stela, taels,tales, teals, tesla]
9: [estrin, inerts, insert, inters, niters, nitres, sinter, triens, trines]
9: [capers, crapes, escarp, pacers, parsec, recaps,scrape, secpar, spacer]
9: [palest, palets, pastel, petals, plates, pleats,septal, staple, tepals]
9: [anestri, antsier, nastier, ratines, retains, retinas,retsina, stainer, stearin]
8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]
8: [aspers, parses, passer, prases, repass, spares,sparse, spears]
8: [enters, nester, renest, rentes, resent, tenser,ternes,treens]
8: [arles, earls, lares, laser, lears, rales, reals, seral]
8: [earings, erasing, gainers, reagins, regains, reginas,searing, seringa]
8: [peris, piers, pries, prise, ripes, speir, spier, spire]
8: [ates, east, eats, etas, sate, seat, seta, teas]
8: [carets, cartes, caster, caters, crates, reacts,recast,traces]
```

Shuffling

The shuffle algorithm does the opposite of what sort does, destroying any trace of order that may have been present in a List. That is, this algorithm reorders the List based on input from a source of randomness such that all possible permutations occur with equal likelihood, assuming a fair source of randomness. This algorithm is useful in implementing games of chance. For example, it could be used to shuffle a List of Card objects representing a deck. Also, it's useful for generating test cases.

This operation has two forms: one takes a List and uses a default source of randomness, and the other requires the caller to provide a Random object to use as a source of randomness. The code for this algorithm is used as an example in the List section.

Routine Data Manipulation

The Collections class provides five algorithms for doing routine data manipulation on List objects, all of which are pretty straightforward:

- reverse — reverses the order of the elements in a List.
- fill — overwrites every element in a List with the specified value. This operation is useful for reinitializing a List.
- copy — takes two arguments, a destination List and a source List, and copies the elements of the source into the destination, overwriting its contents. The destination List must be at least as long as the source. If it is longer, the remaining elements in the destination List are unaffected.
- swap — swaps the elements at the specified positions in a List.

- addAll — adds all the specified elements to a Collection. The elements to be added may be specified individually or as an array.

Searching

The binarySearch algorithm searches for a specified element in a sorted List. This algorithm has two forms.

- Collections.binarySearch(list, key)
- Collections.binarySearch(list, key, comparator)

The return value is the same for both forms.

- If the List contains the search key, its index is returned.
- If not, the return value is (-(insertion point) - 1), where the insertion point is the point at which the value would be inserted into the List, or the index of the first element greater than the value or list.size() if all elements in the List are less than the specified value. This admittedly ugly formula guarantees that the return value will be >= 0 if and only if the search key is found. It's basically a hack to combine a boolean (found) and an integer (index) into a single int return value.

The following idiom, usable with both forms of the binarySearch operation, looks for the specified search key and inserts it at the appropriate position if it's not already present.

```
int pos = Collections.binarySearch(list, key);
if (pos < 0) {
   l.add(-pos-1, key);
   }
```

Composition

The frequency and disjoint algorithms test some aspect of the composition of one or more Collections:

- frequency — counts the number of times the specified element occurs in the specified collection
- disjoint — determines whether two Collections are disjoint; that is, whether they contain no elements in common

Finding Extreme Values

The min and the max algorithms return, respectively, the minimum and maximum element contained in a specified Collection. Both of these operations come in two forms. The simple form takes only a Collection and returns the minimum (or maximum) element according to the elements' natural ordering. The second form takes a Comparator in addition to the Collection and returns the minimum (or maximum) element according to the specified Comparator.

10. Custom Collection Implementations

Reasons to Write an Implementation

- Persistent
- Application-specific
- High-performance, special-purpose
- High-performance, general-purpose
- Enhanced functionality
- Convenience
- Adapter

How to Write a Custom Implementation

The Java Collections Framework provides abstract implementations designed expressly to facilitate custom implementations.

```
public static <T> List<T> asList(T[] a) {
    return new MyArrayList<T>(a);
}

private static class MyArrayList<T> extends AbstractList<T> {
    private final T[] a;
    MyArrayList(T[] array) {
        a = array;
    }

    public T get(int index) {
        return a[index];
    }

    public T set(int index, T element) {
        T oldValue = a[index];
        a[index] = element;
        return oldValue;
    }

    public int size() {
        return a.length;
    }
}
```

Believe it or not, this is very close to the implementation that is contained in `java.util.Arrays`. It's that simple! You provide a constructor and the get, set, and size methods, and AbstractList does all the rest. You get the ListIterator, bulk operations, search operations, hash code computation, comparison, and string representation for free.

Suppose you want to make the implementation a bit faster. The API documentation for abstract implementations describes precisely how each method is implemented, so you'll know which methods to override to get the performance you want. The preceding implementation's performance is fine, but it can be improved a bit. In particular, the toArray method iterates over the List, copying one element at a time. Given the internal representation, it's a lot faster and more sensible just to clone the array.

```
public Object[] toArray() {
    return (Object[]) a.clone();
}
```

With the addition of this override and a few more like it, this implementation is exactly the one found in `java.util.Arrays`. In the interest of full disclosure, it's a bit tougher to use the other abstract implementations because you will have to write your own iterator, but it's still not that difficult.

The following list summarizes the abstract implementations:

- AbstractCollection — a Collection that is neither a Set nor a List. At a minimum, you must provide the iterator and the size methods.
- AbstractSet — a Set; use is identical to AbstractCollection.
- AbstractList — a List backed up by a random-access data store, such as an array. At a minimum, you must provide the positional access methods (get and, optionally, set, remove, and add) and the size method. The abstract class takes care of listIterator (and iterator).
- AbstractSequentialList — a List backed up by a sequential-access data store, such as a linked list. At a minimum, you must provide the listIterator and size methods. The abstract class takes care of the positional access methods. (This is the opposite of AbstractList.)
- AbstractQueue — at a minimum, you must provide the offer, peek, poll, and size methods and an iterator supporting remove.
- AbstractMap — a Map. At a minimum you must provide the entrySet view. This is typically implemented with the AbstractSet class. If the Map is modifiable, you must also provide the put method.

The process of writing a custom implementation follows:

- Choose the appropriate abstract implementation class from the preceding list.
- Provide implementations for all the abstract methods of the class. If your custom collection is to be modifiable, you will have to override one or more of the concrete methods as well. The API documentation for the abstract implementation class will tell you which methods to override.

- Test and, if necessary, debug the implementation. You now have a working custom collection implementation.
- If you are concerned about performance, read the API documentation of the abstract implementation class for all the methods whose implementations you're inheriting. If any seem too slow, override them. If you override any methods, be sure to measure the performance of the method before and after the override. How much effort you put into tweaking performance should be a function of how much use the implementation will get and how critical to performance its use is. (Often this step is best omitted.)

11. Interoperability

- **Compatibility**: This subsection describes how collections can be made to work with older APIs that predate the addition of Collections to the Java platform.
- **API Design**: This subsection describes how to design new APIs so that they will interoperate seamlessly with one another.

11.1 Compatibility

The Java Collections Framework was designed to ensure complete interoperability between the core collection interfaces and the types that were used to represent collections in the early versions of the Java platform: Vector, Hashtable, array, and Enumeration. In this section, you'll learn how to transform old collections to the Java Collections Framework collections and vice versa.

Upward Compatibility

Suppose that you're using an API that returns legacy collections in tandem with another API that requires objects implementing the collection interfaces. To make the two APIs interoperate smoothly, you'll have to transform the legacy collections into modern collections. Luckily, the Java Collections Framework makes this easy.

Suppose the old API returns an array of objects and the new API requires a Collection. The Collections Framework has a convenience implementation that allows an array of objects to be viewed as a List. You use Arrays.asList to pass an array to any method requiring a Collection or a List.

```
Foo[] result = oldMethod(arg);
newMethod(Arrays.asList(result));
```

If the old API returns a Vector or a Hashtable, you have no work to do at all because Vector was retrofitted to implement the List interface, and Hashtable was retrofitted to implement Map. Therefore, a Vector may be passed directly to any method calling for a Collection or a List.

```
Vector result = oldMethod(arg);
newMethod(result);
```

Similarly, a Hashtable may be passed directly to any method calling for a Map.

```
Hashtable result = oldMethod(arg);
newMethod(result);
```

Less frequently, an API may return an Enumeration that represents a collection of objects. The Collections.list method translates an Enumeration into a Collection.

```
Enumeration e = oldMethod(arg);
newMethod(Collections.list(e));
```

Backward Compatibility

Suppose you're using an API that returns modern collections in tandem with another API that requires you to pass in legacy collections. To make the two APIs interoperate smoothly, you have to transform modern collections into old collections. Again, the Java Collections Framework makes this easy.

Suppose the new API returns a Collection, and the old API requires an array of Object. As you're probably aware, the Collection interface contains a toArray method designed expressly for this situation.

```
Collection c = newMethod();
oldMethod(c.toArray());
```

What if the old API requires an array of String (or another type) instead of an array of Object? You just use the other form of toArray — the one that takes an array on input.

```
Collection c = newMethod();
oldMethod((String[]) c.toArray(new String[0]));
```

If the old API requires a Vector, the standard collection constructor comes in handy.

```
Collection c = newMethod();
oldMethod(new Vector(c));
```

The case where the old API requires a Hashtable is handled analogously.

```
Map m = newMethod();
oldMethod(new Hashtable(m));
```

Finally, what do you do if the old API requires an Enumeration? This case isn't common, but it does happen from time to time, and the Collections.enumeration method was provided to handle it. This is a static factory method that takes a Collection and returns an Enumeration over the elements of the Collection.

```
Collection c = newMethod();
oldMethod(Collections.enumeration(c));
```

11.2 API Design

In this short but important section, you'll learn a few simple guidelines that will allow your API to interoperate seamlessly with all other APIs that follow these guidelines. In essence, these rules define what it takes to be a good "citizen" in the world of collections.

Parameters

- If your API contains a method that requires a collection on input, it is of paramount importance that you declare the relevant parameter type to be one of the collection interface types. Never use an implementation type because this defeats the purpose of an interface-based Collections Framework, which is to allow collections to be manipulated without regard to implementation details.
- Further, you should always use the least-specific type that makes sense. For example, don't require a List or a Set if a Collection would do. It's not that you should never require a List or a Set on input; it is correct to do so if a method depends on a property of one of these interfaces. For example, many of the algorithms provided by the Java platform require a List on input because they depend on the fact that lists are ordered. As a general rule, however, the best types to use on input are the most general: Collection and Map.
    - **Caution**: Never define your own ad hoc collection class and require objects of this class on input. By doing this, you'd lose all the benefits provided by the Java Collections Framework.

Return Values

- You can afford to be much more flexible with return values than with input parameters. It's fine to return an object of any type that implements or extends one of the collection interfaces. This can be one of the interfaces or a special-purpose type that extends or implements one of these interfaces.
- For example, one could imagine an image-processing package, called ImageList, that returned objects of a new class that implements List. In addition to the List operations, ImageList could support any application-specific operations that seemed desirable. For example, it might provide an indexImage operation that returned an image containing thumbnail images of each graphic in the ImageList. It's critical to note that even if the API furnishes ImageList instances on output, it should accept arbitrary Collection (or perhaps List) instances on input.
- In one sense, return values should have the opposite behavior of input parameters: It's best to return the most specific applicable collection interface rather than the most general. For example, if you're sure that you'll always return a SortedMap, you should give the relevant method the return type of SortedMap rather than Map. SortedMap instances are more time-consuming to build than ordinary Map instances and are also more powerful. Given that your module has already invested the time to build a SortedMap, it makes good sense to give the user access to its increased power. Furthermore, the user will be able to pass the returned object to methods that demand a SortedMap, as well as those that accept any Map.

Legacy APIs

There are currently plenty of APIs out there that define their own ad hoc collection types. While this is unfortunate, it's a fact of life, given that there was no Collections Framework in the first two major releases of the Java platform. Suppose you own one of these APIs; here's what you can do about it.

- If possible, retrofit your legacy collection type to implement one of the standard collection interfaces. Then all the collections you return will interoperate smoothly with other collection-based APIs. If this is impossible (for example, because one or more of the preexisting type signatures conflict with the standard collection interfaces), define an adapter class that wraps one of your legacy collections objects, allowing it to function as a standard collection. (The Adapter class is an example of a custom implementation.)
- Retrofit your API with new calls that follow the input guidelines to accept objects of a standard collection interface, if possible. Such calls can coexist with the calls that take the legacy collection type. If this is impossible, provide a constructor or static factory for

your legacy type that takes an object of one of the standard interfaces and returns a legacy collection containing the same elements (or mappings). Either of these approaches will allow users to pass arbitrary collections into your API.

12. Summary

- Array type collection：read fast.
- List type collection：add and delete fast，keep insert order.
- Hash type collection: override hashcode and equals to maintain uniqueness.
- Tree type collection：can be sorted by Comparable or Comparator.
- All collection type are un-synchronized except these three are synchronized Hashtable (not from collection), Vector, Stack. To make a collection synchronized:

```
Collections.synchronizedCollection(Collection<T> c)
```

Chapter V: Generics

1. Why Generics

Before JDK 1.5, compiler can only guarantee return type is object. To return an Integer, you have to use casting. So it is easy to generate runtime error (`ClassCastException`) consequence.

```
List myIntList = new ArrayList();// line 1
myIntList.add(new Integer(0));// line 2
Integer x = (Integer) myIntList.iterator().next();// cast
```

Generics is designed to solve above drawback. It can restrict the data type of a collection at compile time.

```
List<Integer> myIntList = new ArrayList<Integer>(); // Line 1
myIntList.add(new Integer(0)); // Line 2
Integer x = myIntList.iterator().next(); // no cast
```

- Complied classes do not have <>, in order to be compatible with class loader.
- By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

2. Wildcards

2.1 `?` wildcard

```
public static void  printCollection(Collection<?> c) {
        Iterator<?> itr = c.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
```

2.2 Upper Bounded Wildcards: `? extends E`

`E` is the upper bound of wildcard `?`

```
// ? extends E : accept E and E's son
    public static void test2() {
        ArrayList<Person> al = new ArrayList<Person>();
        al.add(new Person("abc",30));
        al.add(new Person("abc4",34));

        ArrayList<Student> al2 = new ArrayList<Student>();
        al2.add(new Student("stu1",11));
        al2.add(new Student("stu2",22));

        ArrayList<Worker> al3 = new ArrayList<Worker>();
        al3.add(new Worker("work1",30));
        al3.add(new Worker("work2",34));

        printCollection(al);
        printCollection(al2);
        printCollection(al3);
    }

    public static void printCollection(Collection<? extends Person> al) {
        Iterator<? extends Person> it = al.iterator();
        while(it.hasNext()) {
            System.out.println(it.next());
        }
    }
```

2.3 Lower Bounded Wildcard: `? super E`

`E` is the lower bound of wildcard `?`

```
// ? super E: accept E and E's father. Note: Does not accept father's other son.
    public static void test3() {
        ArrayList<Person> al = new ArrayList<Person>();
        al.add(new Person("person_print3",30));
        al.add(new Person("person_print3",34));

        ArrayList<Student> al2 = new ArrayList<Student>();
        al2.add(new Student("student_print3",11));
        al2.add(new Student("student_print3",22));

        printCollection(al);
        printCollection(al2);
    }

    // this printCollection does not accept Worker object
    public static void printCollection(Collection<? super Student> al) {
        Iterator<? super Student> it = al.iterator();
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
```

- ? extends E: it is usually used to add E and E's sub-classes into collection.
- ? super E: it is usually used to get E and E's super-classes from collection.

Chapter VI: Essential Java Classes

Chapter VII: Reflection

```
interface Shape {
    int area(int... params);
}

class Circle implements Shape {

    @Override
    public int area(int...params) {
        return 0;
    }

}

public int getArea(String clazz, int... params) throws Exception {
    Shape shape = (Shape)Class.forName(clazz).newInstance();
    return shape.area(params);
}
```

```
a. Relection is good solution for this problem.
interface Shape {
    void setShape(double[] params);
    double getArea();
}

class Circle implements Shape {
    private double r;

    @Override
    public void setShape(double[] params) {
        this.r = params[0];
    }
    @Override
    public double getArea() {
        return PI * r * r;
    }
}

class Square implements Shape {
    private double length;

    @Override
    public void setShape(double[] params) {
        this.length = params[0];
    }
    @Override
```

```
    public double getArea() {
        return length * length;
    }
}
class Rectangle implements Shape {
    private double length;
    private double height;

    @Override
    public void setShape(double[] params) {
        this.length = params[0];
        this.height = params[1];
    }
    @Override
    public double getArea() {
        return length * height;
    }
}

// To calculate the area of different kinds of geometric shapes
public double getArea(String clazz, double[] params) throws ClassNotFoundException{
    Shape shape = (Shape)Class.forName(clazz).newInstance();
    return shape.setShape(params).getArea();
}

b. class MyComparator implements Comparator<Shape> {

        @Override
        public int compare(Shape s1, Shape s2) {
            return s1.getArea() - s2.getArea();
        }
    }

    List<Shape> shapes = new ArrayList<>();
    double[] params = {1.0, 2.0};
    Shape circle = new Circle();
    circle.setShape(params);

    Shape square = new Square();
    square.setShape(params);

    Shape rec = new Rectangle();
    rec.setShape(params);

    shapes.add(circle);
    shapes.add(square);
    shapes.add(rec);
    Collections.sort(shapes, new MyComparator());
```

- SQL question:

```
-- Relationship between teacher and course is one to many
-- Relationship between course and student is many to many

-- Teach table
teacher_id primary key
first_name
last_name

-- Course table
course_id  primary key
course_name
teacher_id foreign key

-- Student table
student_id primary key
student_name

-- Course_Student mapping table
course_id
student_id


--Output all courses that have more than 100 students registered
select course_id, count(student_id) from course_student group by course_id having coun
t(student_id) > 100;
```

Chapter VIII: Concurrency

1. Processes and Threads

In concurrent programming, there are two basic units of execution: processes and threads. Most implementations of the Java virtual machine run as a single process.

- **Processes**

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

- **Threads**

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the main thread. This thread has the ability to create additional threads.

2. Implement Multithread

- **extends Thread**

```
1. Thread() // create a new thread object
2. Thread(Runnable target) // create a new thread object based on Runnable
3. Thread(Runnable t, String name) // assign a name for thread
4. Thread(String name) // create a new thread object with an assgned name
```

- **implement Runnable**

```
1. Need to @Overrid run()
2. It is good for shared resources scenario.
```

Example:

```
// Extends Thread
public class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            System.out.println("MyThread is running");
        }
    }
}
```

```
// Implement Runnable
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        for(int i = 0; i < 20; i++) {
            System.out.println("MyRunnable is running");
        }
    }
}
```

```
// Test two cases
public class A_StartingThreads {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.run();

        MyRunnable myRunnable = new MyRunnable();
        myRunnable.run();

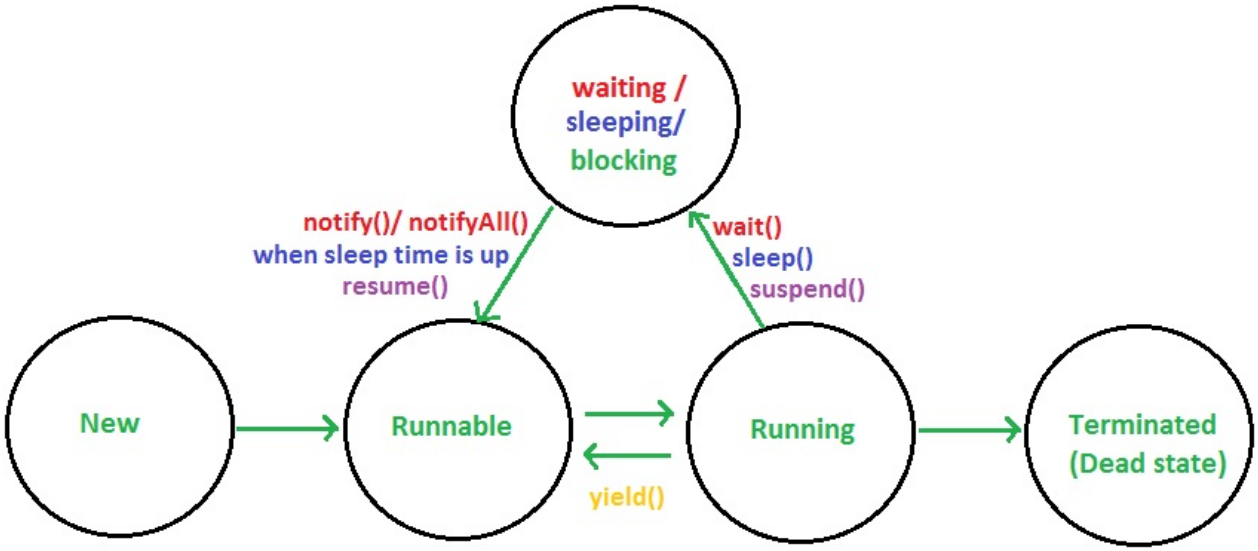
        /*
        * Common pitfall: Calling run() instead of start()
        * */
        Thread thread = new Thread(new MyRunnable());
        thread.start();

        for (int i = 0; i < 10; i++) {
            new Thread("" + i){
                public void run() {
                    System.out.println("Thread: " + getName() + " is running" );
                }
            }.start();
        }
    }
}
```

Summary:

```
Note: Use start() to invoke Thread, not run() method.
* Runnable way is preferable.
  1 When having the Runnable's executed by a thread pool it is easy to queue up the Ru
nnable instances until a thread from the pool is idle.
  2 Sometimes you may have to implement Runnable as well as subclass Thread. For insta
nce, if creating a subclass of Thread that can execute more than one Runnable.
```

3. Thread Life Cycle



Five States:

- NEW
- Runnable
- Running (yield)
- Blocking (wait / sleep / suspend)
- Dead

sleep() method:

```
Thread.sleep(1000)  // sleep 1 second

// interrupt() can wake up the sleep thread
```

join() method:

```
@Override
public void run() {
      // some job
      Thread_out.join();  // current thread will be blocked until this Thread_out comp
letes its job
}
```

isAlive(): it is used to check if current thread is runnable or running.

3. Thread Priority

- Java uses [1 - 10] to setup priority. 10 has the highest priority and 0 has the lowest priority.
- 3 constant thread priority
    - MAX_PRIORITY: 10
    - MIN_PRIORITY: 0
    - NORM_PRIORITY: 5
- Thread.setPriority() / Thread.getPriority()

Note: JVM cannot guarantee that the higher priority thread will run before the lower ones.

4. Thread Synchronization

- **Thread Interference** describes how errors are introduced when multiple threads access shared data.
- **Memory Consistency Errors** describes errors that result from inconsistent views of shared memory.
- **Synchronized Methods** describes a simple idiom that can effectively prevent thread interference and memory consistency errors.
- **Implicit Locks and Synchronization** describes a more general synchronization idiom, and describes how synchronization is based on implicit locks.
- **Atomic Access** talks about the general idea of operations that can't be interfered with by other threads.

4.1 Thread Interference

Consider a simple class called Counter

```
class Counter {
    private int c = 0;
    public void increment() {
        c++;
    }
    public void decrement() {
        c--;
    }
    public int value() {
        return c;
    }
}
```

Interference happens when two operations, running in different threads, but acting on the same data, interleave. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

Note the single expression c++ can be decomposed into three steps:

```
Retrieve the current value of c.
Increment the retrieved value by 1.
Store the incremented value back in c.
```

Suppose Thread A invokes increment at about the same time Thread B invokes decrement. If the initial value of c is 0, their interleaved actions might follow this sequence:

```
1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.
```

Thread A's result is lost, overwritten by Thread B. This particular interleaving is only one possibility. Under different circumstances it might be Thread B's result that gets lost, or there could be no error at all. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

4.2 Memory Consistency Errors

Memory consistency errors occur when different threads have inconsistent views of what should be the same data. The programmer does not need a detailed understanding of these causes. All that is needed is a strategy for avoiding them.

The key to avoiding memory consistency errors is understanding the happens-before relationship. This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement. To see this, consider the following example. Suppose a simple int field is defined and initialized:

```
int counter = 0;
```

The counter field is shared between two threads, A and B. Suppose thread A increments counter:

```
counter++;
```

Then, shortly afterwards, thread B prints out counter:

```
System.out.println(counter);
```

If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be **1**. But if the two statements are executed in separate threads, the value printed out might well be **0**, because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a happens-before relationship between these two statements.

There are several actions that create happens-before relationships. One of them is **synchronization**, as we will see in the following sections.

We've already seen two actions that create happens-before relationships.

- When a statement invokes Thread.start, every statement that has a happens-before relationship with that statement also has a happens-before relationship with every statement executed by the new thread. The effects of the code that led up to the creation of the new thread are visible to the new thread.
- When a thread terminates and causes a Thread.join in another thread to return, then all the statements executed by the terminated thread have a happens-before relationship with all the statements following the successful join. The effects of the code in the thread are now visible to the thread that performed the join.

For a list of actions that create happens-before relationships, refer to the Summary page of the java.util.concurrent package..

4.3 Synchronized Methods

The Java programming language provides two basic synchronization idioms:

- synchronized methods
- synchronized statements

This section is about synchronized methods.

To make a method synchronized, simply add the synchronized keyword to its declaration:

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:

```
* First, it is not possible for two invocations of synchronized methods on the same ob
ject to interleave. When one thread is executing a synchronized method for an object,
all other threads that invoke synchronized methods for the same object block (suspend
execution) until the first thread is done with the object.
* Second, when a synchronized method exits, it automatically establishes a happens-bef
ore relationship with any subsequent invocation of a synchronized method for the same
object. This guarantees that changes to the state of the object are visible to all thr
eads.
```

Note that constructors cannot be synchronized — using the synchronized keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Warning: When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a `List` called instances containing every instance of class. You might be tempted to add the following line to your constructor:

```
instances.add(this);
```

But then other threads can use instances to access the object before construction of the object is complete.

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods. (An important exception: `final` fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed) This strategy is effective, but can present problems with liveness, as we'll see later in this lesson.

4.4 Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the **intrinsic lock** or **monitor lock**. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them. A thread is said to own the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the Class object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

Synchronized Statements

Another way to create synchronized code is with synchronized statements. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

In this example, the `addName` method needs to synchronize changes to `lastName` and `nameCount`, but also needs to avoid synchronizing invocations of other objects' methods. (Invoking other objects' methods from synchronized code can create problems that are described in the section on Liveness.) Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking `nameList.add`.

Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Suppose, for example, class MsLunch has two instance fields, c1 and c2, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of c1 from being interleaved with an update of c2 — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with this, we create two objects solely to provide locks.

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Use this idiom with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.

Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But a thread can acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables reentrant synchronization. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

4.5 Atomic Access

In programming, an atomic action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

We have already seen that an increment expression, such as c++, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, there are actions you can specify that are atomic:

```
* Reads and writes are atomic for **reference variables** and for **most primitive var
iables** (all types except long and double).
* Reads and writes are atomic for **all variables declared volatile** (including long
and double variables).
```

Atomic actions cannot be interleaved, so they can be used without fear of thread interference. However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible. Using `volatile` variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a volatile variable are always visible to other threads. What's more, it also means that when a thread reads a volatile variable, it sees not just the latest change to the volatile, but also the side effects of the code that led up the change.

Using simple atomic variable access is more efficient than accessing these variables through synchronized code, but requires more care by the programmer to avoid memory consistency errors. Whether the extra effort is worthwhile depends on the size and complexity of the application.

Some of the classes in the java.util.concurrent package provide atomic methods that do not rely on synchronization. We'll discuss them in the section on High Level Concurrency Objects.

5. Liveness

- Deadlock
- Starvation and Livelock

5.1 Deadlock

When Deadlock runs, it's extremely likely that both threads will block when they attempt to invoke bowBack. Neither block will ever end, because each thread is waiting for the other to exit bow.

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s"
                + " has bowed back to me!%n",
                this.name, bower.getName());
        }
    }

    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

5.2 Starvation and Livelock

- **Starvation** describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example,

suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

- **Livelock**: A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then livelock may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

6. Guarded Blocks

Threads often have to coordinate their actions. The most common coordination idiom is the guarded block. Such a block begins by polling a condition that must be true before the block can proceed. There are a number of steps to follow in order to do this correctly.

Suppose, for example guardedJoy is a method that must not proceed until a shared variable joy has been set by another thread. Such a method could, in theory, simply loop until the condition is satisfied, but that loop is wasteful, since it executes continuously while waiting.

```
public void guardedJoy() {
    // Simple loop guard. Wastes
    // processor time. Don't do this!
    while(!joy) {}
    System.out.println("Joy has been achieved!");
}
```

A more efficient guard invokes Object.wait to suspend the current thread. The invocation of wait does not return until another thread has issued a notification that some special event may have occurred — though not necessarily the event this thread is waiting for:

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event, which may not
    // be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

Note: Always invoke wait inside a loop that tests for the condition being waited for. Don't assume that the interrupt was for the particular condition you were waiting for, or that the condition is still true.

Like many methods that suspend execution, wait can throw InterruptedException. In this example, we can just ignore that exception — we only care about the value of joy.

Why is this version of guardedJoy synchronized? Suppose d is the object we're using to invoke wait. When a thread invokes d.wait, it must own the intrinsic lock for d — otherwise an error is thrown. Invoking wait inside a synchronized method is a simple way to acquire the intrinsic lock.

When wait is invoked, the thread releases the lock and suspends execution. At some future time, another thread will acquire the same lock and invoke Object.notifyAll, informing all threads waiting on that lock that something important has happened:

```
public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```

Some time after the second thread has released the lock, the first thread reacquires the lock and resumes by returning from the invocation of wait.

Note: There is a second notification method, notify, which wakes up a single thread. Because notify doesn't allow you to specify the thread that is woken up, it is useful only in massively parallel applications — that is, programs with a large number of threads, all doing similar chores. In such an application, you don't care which thread gets woken up.

Let's use guarded blocks to create a Producer-Consumer application. This kind of application shares data between two threads: the producer, that creates the data, and the consumer, that does something with it. The two threads communicate using a shared object.

Coordination is essential: the consumer thread must not attempt to retrieve the data before the producer thread has delivered it, and the producer thread must not attempt to deliver new data if the consumer hasn't retrieved the old data.

In this example, the data is a series of text messages, which are shared through an object of type Drop:

```
public class Drop {
    // Message sent from producer
    // to consumer.
    private String message;
    // True if consumer should wait
    // for producer to send message,
    // false if producer should wait for
    // consumer to retrieve message.
    private boolean empty = true;

    public synchronized String take() {
        // Wait until message is
        // available.
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = true;
        // Notify producer that
        // status has changed.
        notifyAll();
        return message;
    }

    public synchronized void put(String message) {
        // Wait until message has
        // been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = false;
        // Store message.
        this.message = message;
        // Notify consumer that status
        // has changed.
        notifyAll();
    }
}
```

The producer thread, defined in Producer, sends a series of familiar messages. The string "DONE" indicates that all messages have been sent. To simulate the unpredictable nature of real-world applications, the producer thread pauses for random intervals between messages.

```
import java.util.Random;

public class Producer implements Runnable {
    private Drop drop;

    public Producer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        Random random = new Random();

        for (int i = 0;
             i < importantInfo.length;
             i++) {
            drop.put(importantInfo[i]);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
        drop.put("DONE");
    }
}
```

The consumer thread, defined in Consumer, simply retrieves the messages and prints them out, until it retrieves the "DONE" string. This thread also pauses for random intervals.

```
import java.util.Random;

public class Consumer implements Runnable {
    private Drop drop;

    public Consumer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        Random random = new Random();
        for (String message = drop.take();
             ! message.equals("DONE");
             message = drop.take()) {
            System.out.format("MESSAGE RECEIVED: %s%n", message);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
    }
}
```

Finally, here is the main thread, defined in ProducerConsumerExample, that launches the producer and consumer threads.

```
public class ProducerConsumerExample {
    public static void main(String[] args) {
        Drop drop = new Drop();
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}
```

Note: The Drop class was written in order to demonstrate guarded blocks. To avoid re-inventing the wheel, examine the existing data structures in the Java Collections Framework before trying to code your own data-sharing objects. For more information, refer to the Questions and Exercises section.

7. Immutable Objects

An object is considered immutable if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

Programmers are often reluctant to employ immutable objects, because they worry about the cost of creating a new object as opposed to updating an object in place. The impact of object creation is often overestimated, and can be offset by some of the efficiencies associated with immutable objects. These include decreased overhead due to garbage collection, and the elimination of code needed to protect mutable objects from corruption.

The following subsections take a class whose instances are mutable and derives a class with immutable instances from it. In so doing, they give general rules for this kind of conversion and demonstrate some of the advantages of immutable objects.

7.1 A Synchronized Class Example

```
public class SynchronizedRGB {

    // Values must be between 0 and 255.
    private int red;
    private int green;
    private int blue;
    private String name;

    private void check(int red,int green,int blue) {
        if (red < 0 || red > 255
            || green < 0 || green > 255
            || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public SynchronizedRGB(int red,int green,int blue,String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }

    public void set(int red,int green,int blue, String name) {
        check(red, green, blue);
        synchronized (this) {
            this.red = red;
            this.green = green;
            this.blue = blue;
            this.name = name;
        }
    }

    public synchronized int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public synchronized String getName() {
        return name;
    }

    public synchronized void invert() {
        red = 255 - red;
        green = 255 - green;
        blue = 255 - blue;
        name = "Inverse of " + name;
    }
}
```

SynchronizedRGB must be used carefully to avoid being seen in an inconsistent state. Suppose, for example, a thread executes the following code:

```
SynchronizedRGB color = new SynchronizedRGB(0, 0, 0, "Pitch Black");
...
int myColorInt = color.getRGB();      //Statement 1
String myColorName = color.getName(); //Statement 2
```

If another thread invokes color.set after Statement 1 but before Statement 2, the value of myColorInt won't match the value of myColorName. To avoid this outcome, the two statements must be bound together:

```
synchronized (color) {
    int myColorInt = color.getRGB();
    String myColorName = color.getName();
}
```

This kind of inconsistency is only possible for mutable objects — it will not be an issue for the immutable version of SynchronizedRGB.

7.2 A Strategy for Defining Immutable Objects

The following rules define a simple strategy for creating immutable objects. Not all classes documented as "immutable" follow these rules. This does not necessarily mean the creators of these classes were sloppy — they may have good reason for believing that instances of their classes never change after construction. However, such strategies require sophisticated analysis and are not for beginners.

- Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
- Make all fields `final` and `private`.
- Don't allow subclasses to override methods. The simplest way to do this is to declare the class as `final`. A more sophisticated approach is to make the constructor `private` and construct instances in factory methods.
- If the instance fields include references to mutable objects, don't allow those objects to be changed:

```
   * Don't provide methods that modify the mutable objects.
   * Don't share references to the mutable objects. Never store references to exter
nal, mutable objects passed to the constructor; if necessary, create copies, and s
tore references to the copies. Similarly, create copies of your internal mutable o
bjects when necessary to avoid returning the originals in your methods.
```

Applying this strategy to SynchronizedRGB results in the following steps:

- There are two setter methods in this class. The first one, set, arbitrarily transforms the object, and has no place in an immutable version of the class. The second one, invert, can be adapted by having it create a new object instead of modifying the existing one.
- All fields are already private; they are further qualified as final.
- The class itself is declared final.
- Only one field refers to an object, and that object is itself immutable. Therefore, no safeguards against changing the state of "contained" mutable objects are necessary.

```
final public class ImmutableRGB {

    // Values must be between 0 and 255.
    final private int red;
    final private int green;
    final private int blue;
    final private String name;

    private void check(int red,int green,int blue) {
        if (red < 0 || red > 255
            || green < 0 || green > 255
            || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public ImmutableRGB(int red,int green,int blue,String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }

    public int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public String getName() {
        return name;
    }

    public ImmutableRGB invert() {
        return new ImmutableRGB(255 - red,
                        255 - green,
                        255 - blue,
                        "Inverse of " + name);
    }
}
```

8. High Level Concurrency Objects

Most of these features are implemented in the new `java.util.concurrent` packages. There are also new concurrent data structures in the Java Collections Framework.

- **Lock objects:** support locking idioms that simplify many concurrent applications.
- **Executors:** define a high-level API for launching and managing threads.
- **Concurrent collections:** make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
- **Atomic variables:** have features that minimize synchronization and help avoid memory consistency errors.
- **ThreadLocalRandom** (in JDK 7) provides efficient generation of pseudorandom numbers from multiple threads.

8.1 Lock Objects

Synchronized code relies on a simple kind of reentrant lock. This kind of lock is easy to use, but has many limitations. More sophisticated locking idioms are supported by the java.util.concurrent.locks package.

`Lock` objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a Lock object at a time. Lock objects also support a wait/notify mechanism, through their associated Condition objects.

The biggest advantage of Lock objects over implicit locks is their ability to back out of an attempt to acquire a lock. The `tryLock` method backs out if the lock is not available immediately or before a timeout expires (if specified). The `lockInterruptibly` method backs out if another thread sends an interrupt before the lock is acquired.

Let's use Lock objects to solve the deadlock problem we saw in Liveness. Alphonse and Gaston have trained themselves to notice when a friend is about to bow. We model this improvement by requiring that our Friend objects must acquire locks for both participants before proceeding with the bow. Here is the source code for the improved model, Safelock. To demonstrate the versatility of this idiom, we assume that Alphonse and Gaston are so infatuated with their newfound ability to bow safely that they can't stop bowing to each other:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;

public class Safelock {
    static class Friend {
        private final String name;
        private final Lock lock = new ReentrantLock();
```

```
    public Friend(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public boolean impendingBow(Friend bower) {
        Boolean myLock = false;
        Boolean yourLock = false;
        try {
            myLock = lock.tryLock();
            yourLock = bower.lock.tryLock();
        } finally {
            if (! (myLock && yourLock)) {
                if (myLock) {
                    lock.unlock();
                }
                if (yourLock) {
                    bower.lock.unlock();
                }
            }
        }
        return myLock && yourLock;
    }

    public void bow(Friend bower) {
        if (impendingBow(bower)) {
            try {
                System.out.format("%s: %s has"
                    + " bowed to me!%n",
                    this.name, bower.getName());
                bower.bowBack(this);
            } finally {
                lock.unlock();
                bower.lock.unlock();
            }
        } else {
            System.out.format("%s: %s started"
                + " to bow to me, but saw that"
                + " I was already bowing to"
                + " him.%n",
                this.name, bower.getName());
        }
    }

    public void bowBack(Friend bower) {
        System.out.format("%s: %s has" +
            " bowed back to me!%n",
            this.name, bower.getName());
    }
```

```
    }

    static class BowLoop implements Runnable {
        private Friend bower;
        private Friend bowee;

        public BowLoop(Friend bower, Friend bowee) {
            this.bower = bower;
            this.bowee = bowee;
        }

        public void run() {
            Random random = new Random();
            for (;;) {
                try {
                    Thread.sleep(random.nextInt(10));
                } catch (InterruptedException e) {}
                bowee.bow(bower);
            }
        }
    }

    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new BowLoop(alphonse, gaston)).start();
        new Thread(new BowLoop(gaston, alphonse)).start();
    }
}
```

8.2 Executors

In all of the previous examples, there's a close connection between the task being done by a new thread, as defined by its Runnable object, and the thread itself, as defined by a Thread object. This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as executors. The following subsections describe executors in detail.

- **Executor Interfaces** define the three executor object types.
- **Thread Pools** are the most common kind of executor implementation.
- **Fork/Join** is a framework (new in JDK 7) for taking advantage of multiple processors.

8.2.1 Executor Interfaces

The java.util.concurrent package defines three executor interfaces:

Executor, a simple interface that supports launching new tasks. ExecutorService, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself. ScheduledExecutorService, a subinterface of ExecutorService, supports future and/or periodic execution of tasks. Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

The Executor Interface

The Executor interface provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom. If r is a Runnable object, and e is an Executor object you can replace

(new Thread(r)).start(); with

e.execute(r); However, the definition of execute is less specific. The low-level idiom creates a new thread and launches it immediately. Depending on the Executor implementation, execute may do the same thing, but is more likely to use an existing worker thread to run r, or to place r in a queue to wait for a worker thread to become available. (We'll describe worker threads in the section on Thread Pools.)

The executor implementations in java.util.concurrent are designed to make full use of the more advanced ExecutorService and ScheduledExecutorService interfaces, although they also work with the base Executor interface.

The ExecutorService Interface

The ExecutorService interface supplements execute with a similar, but more versatile submit method. Like execute, submit accepts Runnable objects, but also accepts Callable objects, which allow the task to return a value. The submit method returns a Future object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.

ExecutorService also provides methods for submitting large collections of Callable objects. Finally, ExecutorService provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle interrupts correctly.

The ScheduledExecutorService Interface

The ScheduledExecutorService interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified delay. In addition, the interface defines scheduleAtFixedRate and scheduleWithFixedDelay, which executes specified tasks repeatedly, at defined intervals.

8.2.2 Thread Pools

Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

An important advantage of the fixed thread pool is that applications using it degrade gracefully. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to all requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

A simple way to create an executor that uses a fixed thread pool is to invoke the newFixedThreadPool factory method in java.util.concurrent.Executors This class also provides the following factory methods:

The newCachedThreadPool method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks. The newSingleThreadExecutor method creates an executor that executes a single task at a time. Several factory methods are ScheduledExecutorService versions of the above executors. If none of the executors provided by the above factory methods meet your needs, constructing instances of java.util.concurrent.ThreadPoolExecutor or java.util.concurrent.ScheduledThreadPoolExecutor will give you additional options.

8.2.3 Fork/Join

The fork/join framework is an implementation of the ExecutorService interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

As with any ExecutorService implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a **work-stealing algorithm**. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The center of the fork/join framework is the ForkJoinPool class, an extension of the AbstractExecutorService class. ForkJoinPool implements the core work-stealing algorithm and can execute ForkJoinTask processes.

Basic Use

The first step for using the fork/join framework is to write code that performs a segment of the work. Your code should look similar to the following pseudocode:

if (my portion of the work is small enough) do the work directly else split my work into two pieces invoke the two pieces and wait for the results Wrap this code in a ForkJoinTask subclass, typically using one of its more specialized types, either RecursiveTask (which can return a result) or RecursiveAction.

After your ForkJoinTask subclass is ready, create the object that represents all the work to be done and pass it to the invoke() method of a ForkJoinPool instance.

Blurring for Clarity

To help you understand how the fork/join framework works, consider the following example. Suppose that you want to blur an image. The original source image is represented by an array of integers, where each integer contains the color values for a single pixel. The blurred destination image is also represented by an integer array with the same size as the source.

Performing the blur is accomplished by working through the source array one pixel at a time. Each pixel is averaged with its surrounding pixels (the red, green, and blue components are averaged), and the result is placed in the destination array. Since an image is a large array, this process can take a long time. You can take advantage of concurrent processing on multiprocessor systems by implementing the algorithm using the fork/join framework. Here is one possible implementation:

public class ForkBlur extends RecursiveAction { private int[] mSource; private int mStart; private int mLength; private int[] mDestination;

```
// Processing window size; should be odd.
private int mBlurWidth = 15;

public ForkBlur(int[] src, int start, int length, int[] dst) {
    mSource = src;
    mStart = start;
    mLength = length;
    mDestination = dst;
}

protected void computeDirectly() {
    int sidePixels = (mBlurWidth - 1) / 2;
    for (int index = mStart; index < mStart + mLength; index++) {
        // Calculate average.
        float rt = 0, gt = 0, bt = 0;
        for (int mi = -sidePixels; mi <= sidePixels; mi++) {
            int mindex = Math.min(Math.max(mi + index, 0),
                                  mSource.length - 1);
            int pixel = mSource[mindex];
            rt += (float)((pixel & 0x00ff0000) >> 16)
                  / mBlurWidth;
            gt += (float)((pixel & 0x0000ff00) >>  8)
                  / mBlurWidth;
            bt += (float)((pixel & 0x000000ff) >>  0)
                  / mBlurWidth;
        }

        // Reassemble destination pixel.
        int dpixel = (0xff000000     ) |
               (((int)rt) << 16) |
               (((int)gt) <<  8) |
               (((int)bt) <<  0);
        mDestination[index] = dpixel;
    }
}
```

... Now you implement the abstract compute() method, which either performs the blur directly or splits it into two smaller tasks. A simple array length threshold helps determine whether the work is performed or split.

protected static int sThreshold = 100000;

protected void compute() { if (mLength < sThreshold) { computeDirectly(); return; }

```
int split = mLength / 2;

invokeAll(new ForkBlur(mSource, mStart, split, mDestination),
          new ForkBlur(mSource, mStart + split, mLength - split,
                       mDestination));
```

} If the previous methods are in a subclass of the RecursiveAction class, then setting up the task to run in a ForkJoinPool is straightforward, and involves the following steps:

Create a task that represents all of the work to be done.

// source image pixels are in src // destination image pixels are in dst ForkBlur fb = new ForkBlur(src, 0, src.length, dst); Create the ForkJoinPool that will run the task.

ForkJoinPool pool = new ForkJoinPool(); Run the task.

pool.invoke(fb); For the full source code, including some extra code that creates the destination image file, see the ForkBlur example.

Standard Implementations

Besides using the fork/join framework to implement custom algorithms for tasks to be performed concurrently on a multiprocessor system (such as the ForkBlur.java example in the previous section), there are some generally useful features in Java SE which are already implemented using the fork/join framework. One such implementation, introduced in Java SE 8, is used by the java.util.Arrays class for its parallelSort() methods. These methods are similar to sort(), but leverage concurrency via the fork/join framework. Parallel sorting of large arrays is faster than sequential sorting when run on multiprocessor systems. However, how exactly the fork/join framework is leveraged by these methods is outside the scope of the Java Tutorials. For this information, see the Java API documentation.

Another implementation of the fork/join framework is used by methods in the java.util.streams package, which is part of Project Lambda scheduled for the Java SE 8 release. For more information, see the Lambda Expressions section.

8.3 Concurrent Collections

The java.util.concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

BlockingQueue defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue. ConcurrentMap is a subinterface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap. ConcurrentNavigableMap is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is ConcurrentSkipListMap, which is a concurrent analog of

TreeMap. All of these collections help avoid Memory Consistency Errors by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

8.4 Atomic Variables

The java.util.concurrent.atomic package defines classes that support atomic operations on single variables. All classes have get and set methods that work like reads and writes on volatile variables. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables.

To see how this package might be used, let's return to the Counter class we originally used to demonstrate thread interference:

class Counter { private int c = 0;

```
public void increment() {
    c++;
}

public void decrement() {
    c--;
}

public int value() {
    return c;
}
```

} One way to make Counter safe from thread interference is to make its methods synchronized, as in SynchronizedCounter:

class SynchronizedCounter { private int c = 0;

```
public synchronized void increment() {
    c++;
}

public synchronized void decrement() {
    c--;
}

public synchronized int value() {
    return c;
}
```

} For this simple class, synchronization is an acceptable solution. But for a more complicated class, we might want to avoid the liveness impact of unnecessary synchronization. Replacing the int field with an AtomicInteger allows us to prevent thread interference without resorting to synchronization, as in AtomicCounter:

import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter { private AtomicInteger c = new AtomicInteger(0);

```
public void increment() {
    c.incrementAndGet();
}

public void decrement() {
    c.decrementAndGet();
}

public int value() {
    return c.get();
}
```

}

8.5 Concurrent Random Numbers

In JDK 7, java.util.concurrent includes a convenience class, ThreadLocalRandom, for applications that expect to use random numbers from multiple threads or ForkJoinTasks.

For concurrent access, using ThreadLocalRandom instead of Math.random() results in less contention and, ultimately, better performance.

All you need to do is call ThreadLocalRandom.current(), then call one of its methods to retrieve a random number. Here is one example:

int r = ThreadLocalRandom.current() .nextInt(4, 77);

Synchronization Questions

Q: What is synchronization and why is it important?

```
A: With respect to multithreading, synchronization is the capability to control the ac
cess of multiple threads to shared resources. Without synchronization, it is possible
for one thread to modify a shared object while another thread is in the process of usi
ng or updating that object's value. This often leads to significant errors.
```

Q: What are synchronized methods and synchronized statements?

```
1): Synchronized methods are methods that are used to control access to an object. A t
hread only executes a synchronized method after it has acquired the lock for the metho
d's object or class.

2): Synchronized statements are similar to synchronized methods. A synchronized statem
ent can only be executed after a thread has acquired the lock for the object or class
referenced in the synchronized statement.
```

Q: When a thread is created and started, what is its initial state?

```
A: A thread is in the ready state after it has been created and started.
```

Q: What is daemon thread and which method is used to create the daemon thread?

```
A: Daemon thread is a low priority thread which runs intermittently in the back ground
 doing the garbage collection operation for the java runtime system.

setDaemon method is used to create a daemon thread.
```

Q: What method must be implemented by all threads?

```
A: run() method. Whether they
1) extends Thread
2) implements Runnable // this way is preferable. It can extend other class.
```

More Questions

Q: How does Java handle integer overflows and underflows?

```
A: It uses those low order bytes of the result that can fit into the size of the type
allowed by the operation.
```

Q: Does garbage collection guarantee that a program will not run out of memory?

```
A: No.
1) It is possible for programs to use up memory resources faster than they are garbage
 collected.
2) It is also possible for programs to create objects that are not subject to garbage
collection
```

Q: What is the difference between preemptive scheduling and time slicing?

```
1) Under preemptive scheduling, the highest priority task executes until it enters the
 waiting or dead states or a higher priority task comes into existence.

2) Under time slicing, a task executes for a predefined slice of time and then reenter
s the pool of ready tasks. The scheduler then determines which task should execute nex
t, based on priority and other factors.
```

Q: What is the purpose of finalization?

```
A: It is to give an unreachable object the opportunity to perform any cleanup processi
ng before the object is garbage collected.
```

Q: What is the difference between processes and threads?

```
1) A process is an execution of a program
2) A thread is a single execution sequence within the process.
3) A process can contain multiple threads.
4) A thread is sometimes called a lightweight process.
```

Q: Briefly explain high-level thread states?

```
The state chart diagram below describes the thread states.

1) Runnable – A thread becomes runnable when you call the start( ), but does  not nece
ssarily start running immediately.  It will be pooled waiting for its turn to be picke
d for execution by the thread scheduler based on thread priorities.

2) Running: The processor is actively executing the thread code. It runs until it beco
mes blocked, or voluntarily gives up its turn with this static method Thread.yield( ).
 Because of context switching overhead, yield( ) should not be used very frequently

3) Waiting:
a. setup time is optional.
b. release both execution right and lock
c. Needs notify() to wake up.
A thread is in a blocked state while it waits for  some external processing
such as file I/O to finish.A call to currObject.wait( ) method causes the
current thread to wait until some other thread invokescurrObject.notify( )
or the currObject.notifyAll( ) is executed.

4) Sleeping:
a. setup tiem is required.
b. release execution right, but not lock.
Java threads are forcibly put to sleep (suspended) with this overloaded method:
Thread.sleep(milliseconds), Thread.sleep(milliseconds, nanoseconds);

4) Blocked on I/O: Will move to runnable after I/O condition like reading bytes of dat
a etc changes.

5) Blocked on synchronization: will move to running when a lock is acquired.

6) Dead: The thread is finished working.
```

Q. What is the difference between yield and sleeping?

```
1). When a task invokes yield( ), it changes from running state to runnable state.
**Sleep** causes thread to suspend itself for x milliseconds
**Yield** suspends the thread and immediately moves it to the ready queue (the queue w
hich the CPU uses to run threads).

2) When a task invokes sleep ( ), it changes from running state to waiting/sleeping st
ate.
3) The method wait(1000) causes the current thread to wait up to one second a signal f
rom other threads. A thread could wait less than 1 second if it receives the notify( )
 or notifyAll( ) method call.
4) The call to sleep(1000) causes the current thread to sleep for t least 1 second.
```

Java Concurrency Interview Questions

Q.What is atomic operation? What are atomic classes in Java Concurrency API?

```
1) Atomic operations are performed in a single unit of task without interference from
other operations.

2) Atomic operations are necessity in multi-threaded environment to avoid data inconsi
stency.

3) For example  int++ is not an atomic operation. So by the time one threads read it's
 value and increment it by one, other thread has read the older value leading to wrong
 result.

4) To solve this issue, we will have to make sure that increment operation on count is
 atomic, we can do that using Synchronization but Java 5 java.util.concurrent.atomic p
rovides wrapper classes for int and long that can be used to achieve this atomically w
ithout usage of Synchronization.
```

Q.What is Lock interface in Java Concurrency API? What are it's benefits over synchronization?

```
1) Lock interface provide more extensive locking operations than can be obtained using
 synchronized methods and statements.

2) They allow more flexible structuring, may have quite different properties, and may
support multiple associated Condition objects.

3) The advantages of a lock are:
**it's possible to make them fair
**it's possible to make a thread responsive to interruption while waiting on a Lock ob
ject.
**it's possible to try to acquire the lock, but return immediately or after a timeout
if the lock can't be acquired
**it's possible to acquire and release locks in different scopes, and in different ord
ers
```

Q.What is Executors Framework?

```
1) The Executor framework is a framework for standardizing invocation, scheduling, exe
cution, and control of asynchronous tasks according to a set of execution policies.

2) Creating a lot many threads with no bounds to the maximum threshold can cause appli
cation to run out of heap memory. So, creating a ThreadPool is a better solution as a
finite number of threads can be pooled and reused.

3) Executors framework facilitate process of creating Thread pools in java.
```

Q.What is BlockingQueue? How can we implement Producer-Consumer problem using Blocking Queue?

```
1) BlockingQueue is a Queue that supports operations that wait for the queue to become
 non-empty when retrieving and removing an element, and wait for space to become avail
able in the queue when adding an element.

2) BlockingQueue doesn’t accept null values and throw NullPointerException if you try
to store null value in the queue.

3) BlockingQueue implementations are thread-safe. All queuing methods are atomic in na
ture and use internal locks or other forms of concurrency control.

4) BlockingQueue interface is part of java collections framework and it’s primarily us
ed for implementing producer consumer problem.
```

Q.What is Callable and Future?

```
1) Callable interface in concurrency package that is similar to Runnable interface but
 it can return any Object and able to throw Exception.

2) Callable interface use Generic to define the return type of Object. Executors class
 provide useful methods to execute Callable in a thread pool.

3) Since callable tasks run in parallel, we have to wait for the returned Object.

4) Callable tasks return java.util.concurrent.Future object. Using Future we can find
out the status of the Callable task and get the returned Object.

5) It provides get() method that can wait for the Callable to finish and then return t
he result.
```

Q.What is FutureTask Class?

```
1) FutureTask is the base implementation class of Future interface and we can use it w
ith Executors for asynchronous processing.

2) Most of the time we don’t need to use FutureTask class but it comes real handy if w
e want to override some of the methods of Future interface and want to keep most of th
e base implementation. We can just extend this class and override  the methods accordi
ng to our requirements.
```

Q.What are Concurrent Collection Classes?

```
1) Java Collection classes are fail-fast which means that if the Collection will be ch
anged while some thread is traversing over it using iterator, the iterator.next() will
 throw ConcurrentModificationException.

2) Concurrent Collection classes support full concurrency of retrievals and adjustable
 expected concurrency for updates. Major classes are ConcurrentHashMap, CopyOnWriteArr
ayList and CopyOnWriteArraySet
```

Q.What is Executors Class?

```
1) Executors class provide utility methods for Executor, ExecutorService, ScheduledExe
cutorService, ThreadFactory, and Callable classes.

2) Executors class can be used to easily create Thread Pool in java, also this is the
only class supporting execution of Callable implementations.
```

Chapter IX: IO

I/O Streams

- Byte Streams handle I/O of raw binary data.
- Character Streams handle I/O of character data, automatically handling translation to and from the local character set.
- Buffered Streams optimize input and output by reducing the number of calls to the native API.
- Scanning and Formatting allows a program to read and write formatted text.
- I/O from the Command Line describes the Standard Streams and the Console object.
- Data Streams handle binary I/O of primitive data type and String values.
- Object Streams handle binary I/O of objects.

File I/O (Featuring NIO.2)

- What is a Path? examines the concept of a path on a file system.
- The Path Class introduces the cornerstone class of the java.nio.file package.
- Path Operations looks at methods in the Path class that deal with syntactic operations. File Operations introduces concepts common to many of the file I/O methods.
- Checking a File or Directory shows how to check a file's existence and its level of accessibility.
- Deleting a File or Directory.
- Copying a File or Directory.
- Moving a File or Directory.
- Managing Metadata explains how to read and set file attributes.
- Reading, Writing and Creating Files shows the stream and channel methods for reading and writing files.
- Random Access Files shows how to read or write files in a non-sequentially manner.
- Creating and Reading Directories covers API specific to directories, such as how to list a directory's contents.
- Links, Symbolic or Otherwise covers issues specific to symbolic and hard links.
- Walking the File Tree demonstrates how to recursively visit each file and directory in a file tree.
- Finding Files shows how to search for files using pattern matching.
- Watching a Directory for Changes shows how to use the watch service to detect files that are added, removed or updated in one or more directories.

- Other Useful Methods covers important API that didn't fit elsewhere in the lesson.
- Legacy File I/O Code shows how to leverage Path functionality if you have older code using the java.io.File class. A table mapping java.io.File API to java.nio.file API is provided.

Chapter X: Enum and Annotation

Chapter XI: Socket

Lambda Expressions

- Ideal Use Case for Lambda Expressions
  - Approach 1: Create Methods That Search for Members That Match One Characteristic
  - Approach 2: Create More Generalized Search Methods
  - Approach 3: Specify Search Criteria Code in a Local Class
  - Approach 4: Specify Search Criteria Code in an Anonymous Class
  - Approach 5: Specify Search Criteria Code with a Lambda Expression
  - Approach 6: Use Standard Functional Interfaces with Lambda Expressions
  - Approach 7: Use Lambda Expressions Throughout Your Application
  - Approach 8: Use Generics More Extensively
  - Approach 9: Use Aggregate Operations That Accept Lambda Expressions as Parameters
- Lambda Expressions in GUI Applications
- Syntax of Lambda Expressions
- Accessing Local Variables of the Enclosing Scope
- Target Typing
- Target Types and Method Arguments
- Serialization

1. Ideal Use Case for Lambda Expressions

```
public class Person {
    public enum Sex {
        MALE, FEMALE
    }
    String name;
    LocalDate birthday;
    Sex gender;
    String email;

    public int getAge() {
        // ...
    }
    public void printPerson() {
        // ...
    }
}
```

1.1 Approach 1: Create Methods That Search for Members That Match One Characteristic

```
public static void printPersonsOlderThan(List<Person> roster, int age) {
    for (Person p : roster) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

1.2 Approach 2: Create More Generalized Search Methods

```
public static void printPersonsWithinAgeRange(
    List<Person> roster, int low, int high) {
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

What if you want to print members of a specified sex, or a combination of a specified gender and age range? What if you decide to change the Person class and add other attributes such as relationship status or geographical location? Although this method is more generic than `printPersonsOlderThan`, trying to create a separate method for each possible search query can still lead to brittle code. You can instead separate the code that specifies the criteria for which you want to search in a different class.

1.3 Approach 3: Specify Search Criteria Code in a Local Class

The following method prints members that match search criteria that you specify:

```
public static void printPersons(
    List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

This method checks each Person instance contained in the List parameter roster whether it satisfies the search criteria specified in the CheckPerson parameter tester by invoking the method tester.test. If the method tester.test returns a true value, then the method printPersons is invoked on the Person instance.

To specify the search criteria, you implement the CheckPerson interface:

```
interface CheckPerson {
    boolean test(Person p);
}
```

The following class implements the CheckPerson interface by specifying an implementation for the method test. This method filters members that are eligible for Selective Service in the United States: it returns a true value if its Person parameter is male and between the ages of 18 and 25:

```
class CheckPersonEligibleForSelectiveService implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25;
    }
}
```

To use this class, you create a new instance of it and invoke the printPersons method:

```
printPersons(roster, new CheckPersonEligibleForSelectiveService());
```

Although this approach is less brittle—you don't have to rewrite methods if you change the structure of the Person—you still have additional code: a new interface and a local class for each search you plan to perform in your application. Because `CheckPersonEligibleForSelectiveService` implements an interface, you can use an anonymous class instead of a local class and bypass the need to declare a new class for each search.

1.4 Approach 4: Specify Search Criteria Code in an Anonymous Class

One of the arguments of the following invocation of the method printPersons is an anonymous class that filters members that are eligible for Selective Service in the United States: those who are male and between the ages of 18 and 25:

```
printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

This approach reduces the amount of code required because you don't have to create a new class for each search that you want to perform. However, the syntax of anonymous classes is bulky considering that the CheckPerson interface contains only one method. In this case, you can use a lambda expression instead of an anonymous class, as described in the next section.

1.5 Approach 5: Specify Search Criteria Code with a Lambda Expression

The CheckPerson interface is a `functional interface`. A functional interface is any interface that contains only one abstract method. (A functional interface may contain one or more default methods or static methods.) Because a functional interface contains only one abstract method, you can omit the name of that method when you implement it. To do this, instead of using an anonymous class expression, you use a lambda expression, which is highlighted in the following method invocation:

```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

You can use a standard functional interface in place of the interface CheckPerson, which reduces even further the amount of code required.

1.6 Approach 6: Use Standard Functional Interfaces with Lambda Expressions

Reconsider the CheckPerson interface:

```
interface CheckPerson {
    boolean test(Person p);
}
```

This is a very simple interface. It's a `functional interface` because it contains only one abstract method. This method takes one parameter and returns a boolean value. The method is so simple that it might not be worth it to define one in your application. Consequently, the JDK defines several standard functional interfaces, which you can find in the package java.util.function.

For example, you can use the Predicate interface in place of CheckPerson. This interface contains the method boolean test(T t):

```
interface Predicate<T> {
    boolean test(T t);
}
```

The interface Predicate is an example of a generic interface. Generic types (such as generic interfaces) specify one or more type parameters within angle brackets (<>). This interface contains only one type parameter, T. When you declare or instantiate a generic type with actual type arguments, you have a parameterized type. For example, the parameterized type Predicate is the following:

```
interface Predicate<Person> {
    boolean test(Person t);
}
```

This parameterized type contains a method that has the same return type and parameters as CheckPerson.boolean test(Person p). Consequently, you can use Predicate in place of CheckPerson as the following method demonstrates:

```
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

As a result, the following method invocation is the same as when you invoked printPersons in Approach 3: Specify Search Criteria Code in a Local Class to obtain members who are eligible for Selective Service:

```
printPersonsWithPredicate(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

This is not the only possible place in this method to use a lambda expression. The following approach suggests other ways to use lambda expressions.

1.7 Approach 7: Use Lambda Expressions Throughout Your Application

```
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

This method checks each Person instance contained in the List parameter roster whether it satisfies the criteria specified in the Predicate parameter tester. If the Person instance does satisfy the criteria specified by tester, the method printPersron is invoked on the Person instance.

Instead of invoking the method printPerson, you can specify a different action to perform on those Person instances that satisfy the criteria specified by tester. You can specify this action with a lambda expression. Suppose you want a lambda expression similar to printPerson, one that takes one argument (an object of type Person) and returns void. Remember, to use a lambda expression, you need to implement a functional interface. In this case, you need a functional interface that contains an abstract method that can take one argument of type Person and returns void. The Consumer interface contains the method void accept(T t), which has these characteristics. The following method replaces the invocation p.printPerson() with an instance of Consumer that invokes the method accept:

```
public static void processPersons(
    List<Person> roster,
    Predicate<Person> tester,
    Consumer<Person> block) {
        for (Person p : roster) {
            if (tester.test(p)) {
                block.accept(p);
            }
        }
}
```

As a result, the following method invocation is the same as when you invoked printPersons in Approach 3: Specify Search Criteria Code in a Local Class to obtain members who are eligible for Selective Service. The lambda expression used to print members is highlighted:

```
processPersons(
     roster,
     p -> p.getGender() == Person.Sex.MALE
         && p.getAge() >= 18
         && p.getAge() <= 25,
     p -> p.printPerson()
);
```

What if you want to do more with your members' profiles than printing them out. Suppose that you want to validate the members' profiles or retrieve their contact information? In this case, you need a functional interface that contains an abstract method that returns a value. The Function interface contains the method R apply(T t). The following method retrieves the data specified by the parameter mapper, and then performs an action on it specified by the parameter block:

```
public static void processPersonsWithFunction(
    List<Person> roster,
    Predicate<Person> tester,
    Function<Person, String> mapper,
    Consumer<String> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            String data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

The following method retrieves the email address from each member contained in roster who is eligible for Selective Service and then prints it:

```
processPersonsWithFunction(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

1.8 Approach 8: Use Generics More Extensively

Reconsider the method processPersonsWithFunction. The following is a generic version of it that accepts, as a parameter, a collection that contains elements of any data type:

```
public static <X, Y> void processElements(
    Iterable<X> source,
    Predicate<X> tester,
    Function <X, Y> mapper,
    Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

To print the e-mail address of members who are eligible for Selective Service, invoke the processElements method as follows:

```
processElements(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

This method invocation performs the following actions:

- Obtains a source of objects from the collection source. In this example, it obtains a source of Person objects from the collection roster. Notice that the collection roster, which is a collection of type List, is also an object of type Iterable.
- Filters objects that match the Predicate object tester. In this example, the Predicate object is a lambda expression that specifies which members would be eligible for

Selective Service.

- Maps each filtered object to a value as specified by the Function object mapper. In this example, the Function object is a lambda expression that returns the e-mail address of a member.
- Performs an action on each mapped object as specified by the Consumer object block. In this example, the Consumer object is a lambda expression that prints a string, which is the e-mail address returned by the Function object.

You can replace each of these actions with an aggregate operation.

1.9 Approach 9: Use Aggregate Operations That Accept Lambda Expressions as Parameters

The following example uses aggregate operations to print the e-mail addresses of those members contained in the collection roster who are eligible for Selective Service:

```
roster
    .stream()
    .filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```

The following table maps each of the operations the method processElements performs with the corresponding aggregate operation:

`processElements` Action	Aggregate Operation
Obtain a source of objects	`Stream<E> stream()`
Filter objects that match a `Predicate` object	`Stream<T> filter(Predicate<? super T> predicate)`
Map objects to another value as specified by a `Function` object	`<R> Stream<R> map(Function<? super T,? extends R> mapper)`
Perform an action as specified by a `Consumer` object	`void forEach(Consumer<? super T> action)`

The operations filter, map, and forEach are aggregate operations. Aggregate operations process elements from a stream, not directly from a collection (which is the reason why the first method invoked in this example is stream). A stream is a sequence of elements. Unlike a collection, it is not a data structure that stores elements. Instead, a stream carries values from a source, such as collection, through a pipeline. A pipeline is a sequence of stream operations, which in this example is filter- map-forEach. In addition, aggregate operations typically accept lambda expressions as parameters, enabling you to customize how they behave.

For a more thorough discussion of aggregate operations, see the Aggregate Operations lesson.

2. Lambda Expressions in GUI Applications

To process events in a graphical user interface (GUI) application, such as keyboard actions, mouse actions, and scroll actions, you typically create event handlers, which usually involves implementing a particular interface. Often, event handler interfaces are functional interfaces; they tend to have only one method.

In the JavaFX example HelloWorld.java (discussed in the previous section Anonymous Classes), you can replace the highlighted anonymous class with a lambda expression in this statement:

```
        btn.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });
```

The method invocation btn.setOnAction specifies what happens when you select the button represented by the btn object. This method requires an object of type EventHandler. The EventHandler interface contains only one method, void handle(T event). This interface is a functional interface, so you could use the following highlighted lambda expression to replace it:

```
        btn.setOnAction(
          event -> System.out.println("Hello World!")
        );
```

3. Syntax of Lambda Expressions

A lambda expression consists of the following:

- A comma-separated list of formal parameters enclosed in parentheses. The CheckPerson.test method contains one parameter, p, which represents an instance of the Person class.

```
  * **Note**: You can omit the data type of the parameters in a lambda expression.
 In addition, you can omit the parentheses if there is only one parameter. For exa
mple, the following lambda expression is also valid:
```

```
p -> p.getGender() == Person.Sex.MALE
  && p.getAge() >= 18
  && p.getAge() <= 25
```

- The arrow token, ->
- A body, which consists of a single expression or a statement block. This example uses the following expression:

  ```
  p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
  ```

  If you specify a single expression, then the Java runtime evaluates the expression and then returns its value. Alternatively, you can use a return statement:

  ```
  p -> {
    return p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25;
  }
  ```

A return statement is not an expression; in a lambda expression, you must enclose statements in braces ({}). However, you do not have to enclose a void method invocation in braces. For example, the following is a valid lambda expression:

```
email -> System.out.println(email)
```

Note that a lambda expression looks a lot like a method declaration; you can consider lambda expressions as anonymous methods—methods without a name.

The following example, Calculator, is an example of lambda expressions that take more than one formal parameter:

```
public class Calculator {
    interface IntegerMath {
        int operation(int a, int b);
    }

    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {

        Calculator myApp = new Calculator();
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
            myApp.operateBinary(20, 10, subtraction));
    }
}
```

The method operateBinary performs a mathematical operation on two integer operands. The operation itself is specified by an instance of IntegerMath. The example defines two operations with lambda expressions, addition and subtraction. The example prints the following:

```
40 + 2 = 42
20 - 10 = 10
```

4. Target Typing

How do you determine the type of a lambda expression? Recall the lambda expression that selected members who are male and between the ages 18 and 25 years:

```
p -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
```

This lambda expression was used in the following two methods:

- public static void printPersons(List roster, CheckPerson tester) in Approach 3: Specify Search Criteria Code in a Local Class
- public void printPersonsWithPredicate(List roster, Predicate tester) in Approach 6: Use Standard Functional Interfaces with Lambda Expressions

When the Java runtime invokes the method printPersons, it's expecting a data type of CheckPerson, so the lambda expression is of this type. However, when the Java runtime invokes the method printPersonsWithPredicate, it's expecting a data type of Predicate, so the lambda expression is of this type. The data type that these methods expect is called the target type. To determine the type of a lambda expression, the Java compiler uses the target type of the context or situation in which the lambda expression was found. It follows that you can only use lambda expressions in situations in which the Java compiler can determine a target type:

Variable declarations

- Assignments
- Return statements
- Array initializers
- Method or constructor arguments
- Lambda expression bodies
- Conditional expressions, ?:
- Cast expressions

Target Types and Method Arguments

For method arguments, the Java compiler determines the target type with two other language features: overload resolution and type argument inference.

Consider the following two functional interfaces (java.lang.Runnable and java.util.concurrent.Callable):

```
public interface Runnable {
    void run();
}

public interface Callable<V> {
    V call();
}
```

The method Runnable.run does not return a value, whereas Callable.call does.

Suppose that you have overloaded the method invoke as follows (see Defining Methods for more information about overloading methods):

```
void invoke(Runnable r) {
    r.run();
}

<T> T invoke(Callable<T> c) {
    return c.call();
}
```

Which method will be invoked in the following statement?

```
String s = invoke(() -> "done");
```

The method invoke(Callable) will be invoked because that method returns a value; the method invoke(Runnable) does not. In this case, the type of the lambda expression () -> "done" is Callable.

Serialization

You can serialize a lambda expression if its target type and its captured arguments are serializable. However, like inner classes, the serialization of lambda expressions is strongly discouraged.

Aggregate Operations

Note: To better understand the concepts in this section, review the sections Lambda Expressions and Method References.

For what do you use collections? You don't simply store objects in a collection and leave them there. In most cases, you use collections to retrieve items stored in them.

Consider again the scenario described in the section Lambda Expressions. Suppose that you are creating a social networking application. You want to create a feature that enables an administrator to perform any kind of action, such as sending a message, on members of the social networking application that satisfy certain criteria.

```
public class Person {
    public enum Sex {
        MALE, FEMALE
    }
    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;
    // ...
    public int getAge() {
        // ...
    }
    public String getName() {
        // ...
    }
}
```

The following example prints the name of all members contained in the collection roster with a for-each loop:

```
for (Person p : roster) {
    System.out.println(p.getName());
}
```

The following example prints all members contained in the collection roster but with the aggregate operation forEach:

```
roster
    .stream()
    .forEach(e -> System.out.println(e.getName());
```

Although, in this example, the version that uses aggregate operations is longer than the one that uses a for-each loop, you will see that versions that use bulk-data operations will be more concise for more complex tasks.

The following topics are covered:

- Pipelines and Streams
- Differences Between Aggregate Operations and Iterators

1.1 Pipelines and Streams

A pipeline is a sequence of aggregate operations. The following example prints the male members contained in the collection roster with a pipeline that consists of the aggregate operations filter and forEach:

```
roster
    .stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName()));
```

Compare this example to the following that prints the male members contained in the collection roster with a for-each loop:

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());
    }
}
```

A pipeline contains the following components:

```
* A source: This could be a collection, an array, a generator function, or an I/O chan
nel. In this example, the source is the collection roster.
* Zero or more intermediate operations. An intermediate operation, such as filter, pro
duces a new stream.
* A stream is a sequence of elements. Unlike a collection, it is not a data structure
that stores elements. Instead, a stream carries values from a source through a pipelin
e. This example creates a stream from the collection roster by invoking the method str
eam.
* The filter operation returns a new stream that contains elements that match its pred
icate (this operation's parameter). In this example, the predicate is the lambda expre
ssion e -> e.getGender() == Person.Sex.MALE. It returns the boolean value true if the
gender field of object e has the value Person.Sex.MALE. Consequently, the filter opera
tion in this example returns a stream that contains all male members in the collection
 roster.

* A terminal operation. A terminal operation, such as forEach, produces a non-stream r
esult, such as a primitive value (like a double value), a collection, or in the case o
f forEach, no value at all. In this example, the parameter of the forEach operation is
 the lambda expression e -> System.out.println(e.getName()), which invokes the method
getName on the object e. (The Java runtime and compiler infer that the type of the obj
ect e is Person.)
```

The following example calculates the average age of all male members contained in the collection roster with a pipeline that consists of the aggregate operations filter, mapToInt, and average:

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

The mapToInt operation returns a new stream of type IntStream (which is a stream that contains only integer values). The operation applies the function specified in its parameter to each element in a particular stream. In this example, the function is Person::getAge, which is a method reference that returns the age of the member. (Alternatively, you could use the lambda expression e -> e.getAge().) Consequently, the mapToInt operation in this example returns a stream that contains the ages of all male members in the collection roster.

The average operation calculates the average value of the elements contained in a stream of type IntStream. It returns an object of type OptionalDouble. If the stream contains no elements, then the average operation returns an empty instance of OptionalDouble, and invoking the method getAsDouble throws a NoSuchElementException. The JDK contains

many terminal operations such as average that return one value by combining the contents of a stream. These operations are called reduction operations; see the section Reduction for more information.

1.2 Differences Between Aggregate Operations and Iterators

Aggregate operations, like forEach, appear to be like iterators. However, they have several fundamental differences:

- **They use internal iteration**: Aggregate operations do not contain a method like next to instruct them to process the next element of the collection. With internal delegation, your application determines what collection it iterates, but the JDK determines how to iterate the collection. With external iteration, your application determines both what collection it iterates and how it iterates it. However, external iteration can only iterate over the elements of a collection sequentially. Internal iteration does not have this limitation. It can more easily take advantage of parallel computing, which involves dividing a problem into subproblems, solving those problems simultaneously, and then combining the results of the solutions to the subproblems. See the section Parallelism for more information.
- **They process elements from a stream**: Aggregate operations process elements from a stream, not directly from a collection. Consequently, they are also called stream operations.
- **They support behavior as parameters**: You can specify lambda expressions as parameters for most aggregate operations. This enables you to customize the behavior of a particular aggregate operation.

2. Reduction (one of the terminal operations)

Terminal operations:

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach	side effect	to perform a side effect on elements

The section Aggregate Operations describes the following pipeline of operations, which calculates the average age of all male members in the collection roster:

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

The JDK contains many **terminal operations** (such as **average, sum, min, max, and count**) that return one value by combining the contents of a stream. These operations are called **reduction operations**. The JDK also contains reduction operations that return a collection instead of a single value. Many reduction operations perform a specific task, such as finding the average of values or grouping elements into categories. However, the JDK provides you with the general-purpose reduction operations reduce and collect, which this section describes in detail.

This section covers the following topics:

- The Stream.reduce Method
- The Stream.collect Method

2.1 The Stream.reduce Method

The Stream.reduce method is a general-purpose reduction operation. Consider the following pipeline, which calculates the sum of the male members' ages in the collection roster. It uses the Stream.sum reduction operation:

```
Integer totalAge = roster
    .stream()
    .mapToInt(Person::getAge)
    .sum();
```

Compare this with the following pipeline, which uses the Stream.reduce operation to calculate the same value:

```
Integer totalAgeReduce = roster
   .stream()
   .map(Person::getAge)
   .reduce(0,(a, b) -> a + b);
```

The `reduce` operation in this example takes two arguments:

```
    * **identity**: The identity element is both the initial value of the reduction an
d the default result if there are no elements in the stream. In this example, the iden
tity element is 0; this is the initial value of the sum of ages and the default value
if no members exist in the collection roster.

    * **accumulator**: The accumulator function takes two parameters: a partial result
 of the reduction (in this example, the sum of all processed integers so far) and the
next element of the stream (in this example, an integer). It returns a new partial res
ult. In this example, the accumulator function is a lambda expression that adds two In
teger values and returns an Integer value:
```

```
(a, b) -> a + b
```

The `reduce` operation always returns a new value. However, the accumulator function also returns a new value every time it processes an element of a stream. Suppose that you want to reduce the elements of a stream to a more complex object, such as a collection. This might hinder the performance of your application. If your reduce operation involves adding elements to a collection, then every time your accumulator function processes an element, it creates a new collection that includes the element, which is inefficient. It would be more efficient for you to update an existing collection instead. You can do this with the Stream.collect method, which the next section describes.

2.2 The Stream.collect Method

Unlike the reduce method, which always creates a new value when it processes an element, the collect method modifies, or mutates, an existing value.

Consider how to find the average of values in a stream. You require two pieces of data: the total number of values and the sum of those values. However, like the reduce method and all other reduction methods, the collect method returns only one value. You can create a new data type that contains member variables that keep track of the total number of values and the sum of those values, such as the following class, Averager:

```
class Averager implements IntConsumer{
    private int total = 0;
    private int count = 0;

    public double average() {
        return count > 0 ? ((double) total)/count : 0;
    }

    public void accept(int i) { total += i; count++; }
    public void combine(Averager other) {
        total += other.total;
        count += other.count;
    }
}
```

The following pipeline uses the Averager class and the collect method to calculate the average age of all male members:

```
Averager averageCollect = roster.stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(Person::getAge)
    .collect(Averager::new, Averager::accept, Averager::combine);

System.out.println("Average age of male members: " +
    averageCollect.average());
```

The `collect` operation in this example takes three arguments:

```
* **supplier**: The supplier is a factory function; it constructs new instances. For t
he collect operation, it creates instances of the result container. In this example, i
t is a new instance of the Averager class.
* **accumulator**: The accumulator function incorporates a stream element into a resul
t container. In this example, it modifies the Averager result container by incrementin
g the count variable by one and adding to the total member variable the value of the s
tream element, which is an integer representing the age of a male member.
* **combiner**: The combiner function takes two result containers and merges their con
tents. In this example, it modifies an Averager result container by incrementing the c
ount variable by the count member variable of the other Averager instance and adding t
o the total member variable the value of the other Averager instance's total member va
riable.
```

Note the following:

- The supplier is a lambda expression (or a method reference) as opposed to a value like the identity element in the reduce operation.
- The accumulator and combiner functions do not return a value.
- You can use the collect operations with parallel streams; see the section Parallelism for

more information. (If you run the collect method with a parallel stream, then the JDK creates a new thread whenever the combiner function creates a new object, such as an Averager object in this example. Consequently, you do not have to worry about synchronization.)

Although the JDK provides you with the average operation to calculate the average value of elements in a stream, you can use the collect operation and a custom class if you need to calculate several values from the elements of a stream.

The collect operation is best suited for collections. The following example puts the names of the male members in a collection with the collect operation:

```
List<String> namesOfMaleMembersCollect = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(p -> p.getName())
    .collect(Collectors.toList());
```

This version of the collect operation takes one parameter of type Collector. This class encapsulates the functions used as arguments in the collect operation that requires three arguments (supplier, accumulator, and combiner functions).

The Collectors class contains many useful reduction operations, such as accumulating elements into collections and summarizing elements according to various criteria. These reduction operations return instances of the class Collector, so you can use them as a parameter for the collect operation.

This example uses the Collectors.toList operation, which accumulates the stream elements into a new instance of List. As with most operations in the Collectors class, the toList operator returns an instance of Collector, not a collection.

The following example groups members of the collection roster by gender:

```
Map<Person.Sex, List<Person>> byGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(Person::getGender));
```

The groupingBy operation returns a map whose keys are the values that result from applying the lambda expression specified as its parameter (which is called a classification function). In this example, the returned map contains two keys, Person.Sex.MALE and Person.Sex.FEMALE. The keys' corresponding values are instances of List that contain the

stream elements that, when processed by the classification function, correspond to the key value. For example, the value that corresponds to key Person.Sex.MALE is an instance of List that contains all male members.

The following example retrieves the names of each member in the collection roster and groups them by gender:

```
Map<Person.Sex, List<String>> namesByGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(
                Person::getGender,
                Collectors.mapping(
                    Person::getName,
                    Collectors.toList())));
```

The groupingBy operation in this example takes two parameters, a classification function and an instance of Collector. The Collector parameter is called a downstream collector. This is a collector that the Java runtime applies to the results of another collector. Consequently, this groupingBy operation enables you to apply a collect method to the List values created by the groupingBy operator. This example applies the collector mapping, which applies the mapping function Person::getName to each element of the stream. Consequently, the resulting stream consists of only the names of members. A pipeline that contains one or more downstream collectors, like this example, is called a multilevel reduction.

The following example retrieves the total age of members of each gender:

```
Map<Person.Sex, Integer> totalAgeByGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(
                Person::getGender,
                Collectors.reducing(0,
                    Person::getAge,
                    Integer::sum)));
```

The reducing operation takes three parameters:

identity: Like the Stream.reduce operation, the identity element is both the initial value of the reduction and the default result if there are no elements in the stream. In this example, the identity element is 0; this is the initial value of the sum of ages and the default value if no members exist. mapper: The reducing operation applies this mapper function to all stream elements. In this example, the mapper retrieves the age of each member. operation: The

operation function is used to reduce the mapped values. In this example, the operation function adds Integer values. The following example retrieves the average age of members of each gender:

```
Map<Person.Sex, Double> averageAgeByGender = roster
    .stream()
    .collect(
        Collectors.groupingBy(
            Person::getGender,
            Collectors.averagingInt(Person::getAge)));
```

3. Parallelism

Parallel computing involves dividing a problem into subproblems, solving those problems simultaneously (in parallel, with each subproblem running in a separate thread), and then combining the results of the solutions to the subproblems. Java SE provides the fork/join framework, which enables you to more easily implement parallel computing in your applications. However, with this framework, you must specify how the problems are subdivided (partitioned). With aggregate operations, the Java runtime performs this partitioning and combining of solutions for you.

One difficulty in implementing parallelism in applications that use collections is that collections are not thread-safe, which means that multiple threads cannot manipulate a collection without introducing thread interference or memory consistency errors. The Collections Framework provides synchronization wrappers, which add automatic synchronization to an arbitrary collection, making it thread-safe. However, synchronization introduces thread contention. You want to avoid thread contention because it prevents threads from running in parallel. Aggregate operations and parallel streams enable you to implement parallelism with non-thread-safe collections provided that you do not modify the collection while you are operating on it.

Note that parallelism is not automatically faster than performing operations serially, although it can be if you have enough data and processor cores. While aggregate operations enable you to more easily implement parallelism, it is still your responsibility to determine if your application is suitable for parallelism.

This section covers the following topics:

```
* Executing Streams in Parallel
* Concurrent Reduction
* Ordering
* Side Effects
    * Laziness
    * Interference
    * Stateful Lambda Expressions
```

3.1 Executing Streams in Parallel

You can execute streams in serial or in parallel. When a stream executes in parallel, the Java runtime partitions the stream into multiple substreams. Aggregate operations iterate over and process these substreams in parallel and then combine the results.

When you create a stream, it is always a serial stream unless otherwise specified. To create a parallel stream, invoke the operation Collection.parallelStream. Alternatively, invoke the operation BaseStream.parallel. For example, the following statement calculates the average age of all male members in parallel:

```
double average = roster
    .parallelStream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

3.2 Concurrent Reduction

Consider again the following example (which is described in the section Reduction) that groups members by gender. This example invokes the collect operation, which reduces the collection roster into a Map:

```
Map<Person.Sex, List<Person>> byGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(Person::getGender));
The following is the parallel equivalent:

ConcurrentMap<Person.Sex, List<Person>> byGender =
    roster
        .parallelStream()
        .collect(
            Collectors.groupingByConcurrent(Person::getGender));
```

This is called a `concurrent reduction`. The Java runtime performs a concurrent reduction if all of the the following are true for a particular pipeline that contains the collect operation:

```
* The stream is parallel.
* The parameter of the collect operation, the collector, has the characteristic Collec
tor.Characteristics.CONCURRENT. To determine the characteristics of a collector, invok
e the Collector.characteristics method.
* Either the stream is unordered, or the collector has the characteristic Collector.Ch
aracteristics.UNORDERED. To ensure that the stream is unordered, invoke the BaseStream
.unordered operation.
```

Note: This example returns an instance of ConcurrentMap instead of Map and invokes the groupingByConcurrent operation instead of groupingBy. (See the section Concurrent Collections for more information about ConcurrentMap.) Unlike the operation groupingByConcurrent, the operation groupingBy performs poorly with parallel streams. (This is because it operates by merging two maps by key, which is computationally expensive.) Similarly, the operation Collectors.toConcurrentMap performs better with parallel streams than the operation Collectors.toMap.

3.3 Ordering

The order in which a pipeline processes the elements of a stream depends on whether the stream is executed in serial or in parallel, the source of the stream, and intermediate operations. For example, consider the following example that prints the elements of an instance of ArrayList with the forEach operation several times:

```
Integer[] intArray = {1, 2, 3, 4, 5, 6, 7, 8 };
List<Integer> listOfIntegers =
    new ArrayList<>(Arrays.asList(intArray));

System.out.println("listOfIntegers:");
listOfIntegers
    .stream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("listOfIntegers sorted in reverse order:");
Comparator<Integer> normal = Integer::compare;
Comparator<Integer> reversed = normal.reversed();
Collections.sort(listOfIntegers, reversed);
listOfIntegers
    .stream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("Parallel stream");
listOfIntegers
    .parallelStream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("Another parallel stream:");
listOfIntegers
    .parallelStream()
    .forEach(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("With forEachOrdered:");
listOfIntegers
    .parallelStream()
    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");
```

This example consists of five pipelines. It prints output similar to the following:

```
listOfIntegers:
1 2 3 4 5 6 7 8
listOfIntegers sorted in reverse order:
8 7 6 5 4 3 2 1
Parallel stream:
3 4 1 6 2 5 7 8
Another parallel stream:
6 3 1 5 7 8 4 2
With forEachOrdered:
8 7 6 5 4 3 2 1
```

This example does the following:

```
* The first pipeline prints the elements of the list listOfIntegers in the order that
they were added to the list.
* The second pipeline prints the elements of listOfIntegers after it was sorted by the
 method Collections.sort.
* The third and fourth pipelines print the elements of the list in an apparently rando
m order. Remember that stream operations use internal iteration when processing elemen
ts of a stream. Consequently, when you execute a stream in parallel, the Java compiler
 and runtime determine the order in which to process the stream's elements to maximize
 the benefits of parallel computing unless otherwise specified by the stream operation
.
* The fifth pipeline uses the method forEachOrdered, which processes the elements of t
he stream in the order specified by its source, regardless of whether you executed the
 stream in serial or parallel. Note that you may lose the benefits of parallelism if y
ou use operations like forEachOrdered with parallel streams.
```

3.4 Side Effects

A method or an expression has a side effect if, in addition to returning or producing a value, it also modifies the state of the computer. Examples include mutable reductions (operations that use the collect operation; see the section Reduction for more information) as well as invoking the System.out.println method for debugging. The JDK handles certain side effects in pipelines well. In particular, the collect method is designed to perform the most common stream operations that have side effects in a parallel-safe manner. Operations like forEach and peek are designed for side effects; a lambda expression that returns void, such as one that invokes System.out.println, can do nothing but have side effects. Even so, you should use the forEach and peek operations with care; if you use one of these operations with a parallel stream, then the Java runtime may invoke the lambda expression that you specified as its parameter concurrently from multiple threads. In addition, never pass as parameters lambda expressions that have side effects in operations such as filter and map. The following sections discuss interference and stateful lambda expressions, both of which can be sources of side effects and can return inconsistent or unpredictable results, especially in parallel streams. However, the concept of laziness is discussed first, because it has a direct effect on interference.

3.5 Laziness

All intermediate operations are lazy. An expression, method, or algorithm is lazy if its value is evaluated only when it is required. (An algorithm is eager if it is evaluated or processed immediately.) Intermediate operations are lazy because they do not start processing the contents of the stream until the terminal operation commences. Processing streams lazily enables the Java compiler and runtime to optimize how they process streams. For example,

in a pipeline such as the filter-mapToInt-average example described in the section Aggregate Operations, the average operation could obtain the first several integers from the stream created by the mapToInt operation, which obtains elements from the filter operation. The average operation would repeat this process until it had obtained all required elements from the stream, and then it would calculate the average.

3.6 Interference

Lambda expressions in stream operations should not interfere. Interference occurs when the source of a stream is modified while a pipeline processes the stream. For example, the following code attempts to concatenate the strings contained in the List listOfStrings. However, it throws a ConcurrentModificationException:

```
try {
    List<String> listOfStrings =
        new ArrayList<>(Arrays.asList("one", "two"));

    // This will fail as the peek operation will attempt to add the
    // string "three" to the source after the terminal operation has
    // commenced.

    String concatenatedString = listOfStrings
        .stream()

        // Don't do this! Interference occurs here.
        .peek(s -> listOfStrings.add("three"))

        .reduce((a, b) -> a + " " + b)
        .get();

    System.out.println("Concatenated string: " + concatenatedString);

} catch (Exception e) {
    System.out.println("Exception caught: " + e.toString());
}
```

This example concatenates the strings contained in listOfStrings into an Optional value with the reduce operation, which is a terminal operation. However, the pipeline here invokes the intermediate operation peek, which attempts to add a new element to listOfStrings. Remember, all intermediate operations are lazy. This means that the pipeline in this example begins execution when the operation get is invoked, and ends execution when the get operation completes. The argument of the peek operation attempts to modify the stream source during the execution of the pipeline, which causes the Java runtime to throw a ConcurrentModificationException.

3.7 Stateful Lambda Expressions

Avoid using stateful lambda expressions as parameters in stream operations. A stateful lambda expression is one whose result depends on any state that might change during the execution of a pipeline. The following example adds elements from the List listOfIntegers to a new List instance with the map intermediate operation. It does this twice, first with a serial stream and then with a parallel stream:

```
List<Integer> serialStorage = new ArrayList<>();

System.out.println("Serial stream:");
listOfIntegers
    .stream()

    // Don't do this! It uses a stateful lambda expression.
    .map(e -> { serialStorage.add(e); return e; })

    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");

serialStorage
    .stream()
    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");

System.out.println("Parallel stream:");
List<Integer> parallelStorage = Collections.synchronizedList(
    new ArrayList<>());
listOfIntegers
    .parallelStream()

    // Don't do this! It uses a stateful lambda expression.
    .map(e -> { parallelStorage.add(e); return e; })

    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");

parallelStorage
    .stream()
    .forEachOrdered(e -> System.out.print(e + " "));
System.out.println("");
```

The lambda expression e -> { parallelStorage.add(e); return e; } is a stateful lambda expression. Its result can vary every time the code is run. This example prints the following:

```
Serial stream:
8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
Parallel stream:
8 7 6 5 4 3 2 1
1 3 6 2 4 5 8 7
```

The operation forEachOrdered processes elements in the order specified by the stream, regardless of whether the stream is executed in serial or parallel. However, when a stream is executed in parallel, the map operation processes elements of the stream specified by the Java runtime and compiler. Consequently, the order in which the lambda expression e -> { parallelStorage.add(e); return e; } adds elements to the List parallelStorage can vary every time the code is run. For deterministic and predictable results, ensure that lambda expression parameters in stream operations are not stateful.

Note: This example invokes the method synchronizedList so that the List parallelStorage is thread-safe. Remember that collections are not thread-safe. This means that multiple threads should not access a particular collection at the same time. Suppose that you do not invoke the method synchronizedList when creating parallelStorage:

```
List<Integer> parallelStorage = new ArrayList<>();
```

The example behaves erratically because multiple threads access and modify parallelStorage without a mechanism like synchronization to schedule when a particular thread may access the List instance. Consequently, the example could print output similar to the following:

```
Parallel stream:
8 7 6 5 4 3 2 1
null 3 5 4 7 8 1 2
```

JDBC: Java Database Conncetion

1. Processing SQL Statements with JDBC

In general, to process any SQL statement with JDBC, you follow these steps:

```
* Establishing a connection.
* Create a statement.
* Execute the query.
* Process the ResultSet object.
* Close the connection.
```

This page uses the following method, CoffeesTables.viewTable, from the tutorial sample to demonstrate these steps. This method outputs the contents of the table COFFEES. This method will be discussed in more detail later in this tutorial:

```
public static void viewTable(Connection con, String dbName) throws SQLException {
    Statement stmt = null;
    String query = "select COF_NAME, SUP_ID, PRICE, " +
                   "SALES, TOTAL " +
                   "from " + dbName + ".COFFEES";
    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            System.out.println(coffeeName + "\t" + supplierID +
                               "\t" + price + "\t" + sales +
                               "\t" + total);
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

Establishing Connections First, establish a connection with the data source you want to use. A data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver. This connection is represented by a Connection object.

```
public Connection getConnection() throws SQLException {
    Connection conn = null;
    Properties connectionProps = new Properties();
    connectionProps.put("user", this.userName);
    connectionProps.put("password", this.password);

    if (this.dbms.equals("mysql")) {
        conn = DriverManager.getConnection(
                   "jdbc:" + this.dbms + "://" +
                   this.serverName +
                   ":" + this.portNumber + "/",
                   connectionProps);
    } else if (this.dbms.equals("derby")) {
        conn = DriverManager.getConnection(
                   "jdbc:" + this.dbms + ":" +
                   this.dbName +
                   ";create=true",
                   connectionProps);
    }
    System.out.println("Connected to database");
    return conn;
}
```

Creating Statements A Statement is an interface that represents a SQL statement. You execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set. You need a Connection object to create a Statement object.

For example, CoffeesTables.viewTable creates a Statement object with the following code:

```
stmt = con.createStatement();
```

There are three different kinds of statements:

```
* Statement: Used to implement simple SQL statements with **no parameters**.
* PreparedStatement: (Extends Statement.) Used for precompiling SQL statements that mi
ght contain **input parameters**.
* CallableStatement: (Extends PreparedStatement.) Used to execute stored procedures th
at may contain **both input and output parameters**.
```

Executing Queries To execute a query, call an execute method from Statement such as the following:

```
* execute: Returns true if the first object that the query returns is a ResultSet obje
ct. Use this method if the query could return one or more ResultSet objects. Retrieve
the ResultSet objects returned from the query by repeatedly calling Statement.getResul
tSet.
* executeQuery: Returns one ResultSet object.
* executeUpdate: Returns an integer representing the number of rows affected by the SQ
L statement. Use this method if you are using INSERT, DELETE, or UPDATE SQL statements
.
```

For example, CoffeesTables.viewTable executed a Statement object with the following code:

```
ResultSet rs = stmt.executeQuery(query);
```

Processing ResultSet Objects You access the data in a ResultSet object through a cursor. Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in the ResultSet object. Initially, the cursor is positioned before the first row. You call various methods defined in the ResultSet object to move the cursor.

For example, CoffeesTables.viewTable repeatedly calls the method ResultSet.next to move the cursor forward by one row. Every time it calls next, the method outputs the data in the row where the cursor is currently positioned:

```
try {
    stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        int supplierID = rs.getInt("SUP_ID");
        float price = rs.getFloat("PRICE");
        int sales = rs.getInt("SALES");
        int total = rs.getInt("TOTAL");
        System.out.println(coffeeName + "\t" + supplierID +
                           "\t" + price + "\t" + sales +
                           "\t" + total);
    }
}
```

Closing Connections When you are finished using a Statement, call the method Statement.close to immediately release the resources it is using. When you call this method, its ResultSet objects are closed.

For example, the method CoffeesTables.viewTable ensures that the Statement object is closed at the end of the method, regardless of any SQLException objects thrown, by wrapping it in a finally block:

```
} finally {
    if (stmt != null) { stmt.close(); }
}
```

JDBC throws an SQLException when it encounters an error during an interaction with a data source. See Handling SQL Exceptions for more information.

In JDBC 4.1, which is available in Java SE release 7 and later, you can use a try-with-resources statement to automatically close Connection, Statement, and ResultSet objects, regardless of whether an SQLException has been thrown. An automatic resource statement consists of a try statement and one or more declared resources. For example, you can modify CoffeesTables.viewTable so that its Statement object closes automatically, as follows:

```
public static void viewTable(Connection con) throws SQLException {

    String query = "select COF_NAME, SUP_ID, PRICE, " +
                   "SALES, TOTAL " +
                   "from COFFEES";

    try (Statement stmt = con.createStatement()) {

        ResultSet rs = stmt.executeQuery(query);

        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            System.out.println(coffeeName + ", " + supplierID +
                               ", " + price + ", " + sales +
                               ", " + total);
        }
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    }
}
```

The following statement is an try-with-resources statement, which declares one resource, stmt, that will be automatically closed when the try block terminates:

```
try (Statement stmt = con.createStatement()) {
    // ...
}
```

Best Practices in Implementation

Best Practices for Designing the API

1. When deciding on checked exceptions vs. unchecked exceptions, ask yourself, "What action can the client code take when the exception occurs?" If the client can take some alternate action to recover from the exception, make it a checked exception. If the client cannot do anything useful, then make the exception unchecked. By useful, I mean taking steps to recover from the exception and not just logging the exception.

To summarize:
Client's reaction when exception happens
Exception type
Client code cannot do anything
Make it an unchecked exception
Client code will take some useful recovery action based on information in exception
Make it a checked exception

Moreover, prefer unchecked exceptions for all programming errors: unchecked exceptions have the benefit of not forcing the client API to explicitly deal with them. They propagate to where you want to catch them, or they go all the way out and get reported. The Java API has many unchecked exceptions, such as NullPointerException, IllegalArgumentException, and IllegalStateException. I prefer working with standard exceptions provided in Java rather than creating my own. They make my code easy to understand and avoid increasing the memory footprint of code.

1. Preserve encapsulation (for checked exception). Never let implementation-specific checked exceptions escalate to the higher layers. For example, do not propagate SQLException from data access code to the business objects layer. Business objects layer do not need to know about SQLException. You have two options: Convert SQLException into another checked exception, if the client code is expected to recuperate from the exception. Convert SQLException into an unchecked exception, if the client code cannot do anything about it. Most of the time, client code cannot do anything about SQLExceptions. Do not hesitate to convert them into unchecked exceptions. Consider the following piece of code: public void dataAccessCode(){ try{

```
..some code that throws SQLException
```

}catch(SQLException ex){

```
    ex.printStacktrace();
```

}}

This catch block just suppresses the exception and does nothing. The justification is that there is nothing my client could do about an SQLException. How about dealing with it in the following manner?

```
public void dataAccessCode(){
    try{
        ..some code that throws SQLException
    }catch(SQLException ex){
        throw new RuntimeException(ex);
    }
}
```

This converts SQLException to RuntimeException. If SQLException occurs, the catch clause throws a new RuntimeException. The execution thread is suspended and the exception gets reported. However, I am not corrupting my business object layer with unnecessary exception handling, especially since it cannot do anything about an SQLException. If my catch needs the root exception cause, I can make use of the getCause() method.

If you are confident that the business layer can take some recovery action when SQLException occurs, you can convert it into a more meaningful checked exception. But I have found that just throwing RuntimeExceptionsuffices most of the time

1. Try not to create new custom exceptions if they do not have useful information for client code. What is wrong with following code?

```
public class DuplicateUsernameException
 extends Exception {}
```

It is not giving any useful information to the client code, other than an indicative exception name. Do not forget that Java Exception classes are like other classes, wherein you can add methods that you think the client code will invoke to get more information.
We could add useful methods to DuplicateUsernameException, such as:

```
public class DuplicateUsernameException
    extends Exception {
    public DuplicateUsernameException
        (String username){....}
    public String requestedUsername(){...}
    public String[] availableNames(){...}
}
```

The new version provides two useful methods: requestedUsername(), which returns the requested name, and availableNames(), which returns an array of available usernames similar to the one requested. The client could use these methods to inform that the requested username is not available and that other usernames are available. But if you are not going to add extra information, then just throw a standard exception:

throw new Exception("Username already taken");

Even better, if you think the client code is not going to take any action other than logging if the username is already taken, throw a unchecked exception:

throw new RuntimeException("Username already taken");

Alternatively, you can even provide a method that checks if the username is already taken. It is worth repeating that checked exceptions are to be used in situations where the client API can take some productive action based on the information in the exception. Prefer unchecked exceptions for all programmatic errors. They make your code more readable.

1. Document exceptions. You can use Javadoc's @throws tag to document both checked and unchecked exceptions that your API throws. However, I prefer to write unit tests to document exceptions. Tests allow me to see the exceptions in action and hence serve as documentation that can be executed. Whatever you do, have some way by which the client code can learn of the exceptions that your API throws. Here is a sample unit test that tests forIndexOutOfBoundsException:

```
public void testIndexOutOfBoundsException() {
ArrayList blankList = new ArrayList();
try {
blankList.get(10);
fail("Should raise an IndexOutOfBoundsException");
} catch (IndexOutOfBoundsException success) {}
}
```

The code above should throw an IndexOutOfBoundsException when blankList.get(10) is invoked. If it does not, the fail("Should raise an IndexOutOfBoundsException") statement explicitly fails the test. By writing unit tests for exceptions, you not only document how the exceptions work, but also make your code robust by testing for exceptional scenarios.
Best Practices for Using Exceptions

The next set of best practices show how the client code should deal with an API that throws checked exceptions.

1. Always clean up after yourself If you are using resources like database connections or network connections, make sure you clean them up. If the API you are invoking uses

only unchecked exceptions, you should still clean up resources after use, with try -finally blocks.

```
public void dataAccessCode(){
    Connection conn = null;
    try{
        conn = getConnection();
        ..some code that throws SQLException
    }catch(SQLException ex){
        ex.printStacktrace();
    } finally{
        DBUtil.closeConnection(conn);
    }
}

class DBUtil{
    public static void closeConnection
        (Connection conn){
        try{
            conn.close();
        } catch(SQLException ex){
            logger.error("Cannot close connection");
            throw new RuntimeException(ex);
        }
    }
}
```

DBUtil is a utility class that closes the Connection. The important point is the use of finally block, which executes whether or not an exception is caught. In this example, the finally closes the connection and throws aRuntimeException if there is problem with closing the connection.

1. Never use exceptions for flow control Generating stack traces is expensive and the value of a stack trace is in debugging. In a flow-control situation, the stack trace would be ignored, since the client just wants to know how to proceed. In the code below, a custom exception, MaximumCountReachedException, is used to control the flow.

```
public void useExceptionsForFlowControl() {
    try {
        while (true) {
            increaseCount();
        }
    } catch (MaximumCountReachedException ex) {
    }
    //Continue execution
}

public void increaseCount()
    throws MaximumCountReachedException {
    if (count >= 5000)
        throw new MaximumCountReachedException();
}
```

The useExceptionsForFlowControl() uses an infinite loop to increase the count until the exception is thrown. This not only makes the code difficult to read, but also makes it slower. Use exception handling only in exceptional situations.

1. Do not suppress or ignore exceptions When a method from an API throws a checked exception, it is trying to tell you that you should take some counter action. If the checked exception does not make sense to you, do not hesitate to convert it into an unchecked exception and throw it again, but do not ignore it by catching it with {} and then continue as if nothing had happened.
2. Do not catch top-level exceptions Unchecked exceptions inherit from the RuntimeException class, which in turn inherits from Exception. By catching the Exception class, you are also catching RuntimeException as in the following code:

```
try{
..
}catch(Exception ex){
}
```

The code above ignores unchecked exceptions, as well.

1. Always Log exceptions(just once) Logging the same exception stack trace more than once can confuse the programmer examining the stack trace about the original source of exception. So just log it once.

\

file path use "/" instead of "\", so the project becomes plastform independent.

Java Clean Code Practices:

1) Use of descriptive and meaningful variable,**method and class names**as opposed to relying too much on comments. E.g. calculateTax(BigDecimal amount), UserDAOImpl.java, etc.

Bad: List list;

Good: List<String> accounts;

2) Class and functions should be small and focus on doing one thing.Do not duplicate the code.

Example: StudentDao.java for data access logic only, Customer.java for model/entity object, StudentService.java for business logic, and CustomerValidator.java for validating input fields, etc.

Similarly, separate functions like calculateSalary(Long employeeId) will invoke other sub functions with meaningful names like

calculateBonus(Long employeeId),
calculateLeaves(Long employeeId), etc

3) Methods should not take too many parameters.

Bad: processOrder(String id, String name, String address, BigDecimal price, int quantity, BigDecimal discount);

Good: processOrder(CustomerDetail customer, OrderDetail order);

where CustomerDetail is a value object with attributes like code, name etc.

4) Use a standard code formatting template.

Share the template across the development team.

5) D**eclare the variables with the lowest possible scope.**

For example, if a variable “temp” is used only inside a loop, then declare it inside the loop, and not outside.

6) Don’t create variables that you don’t use again.

Example: instead of Customer customer = repository.save(customer);

return customer;

Use: return repository.save(customer);

7) Delete needless and commented out code.

You have source control for the history.

8) Do not use System.out.println statements.

Use proper logging frameworks like slf4j and logback for logging.

9) Make a class final and the object immutable where possible.

Immutable classes are thread-safe and more secured.

For example, the Java String class is immutable and declared as final.

10) Minimize the accessibility of the packages, classes and its members like methods and variables.

Example:private, protected, default, and public access modifiers.

Code to interface as opposed to implementation.

Bad: ArrayList<String> names = new ArrayList<String>();

Good: List<String> names = new ArrayList<String>();

11) Use right data types.

For example, use BigDecimal instead of floating point variables like float or double for monetary values. Use enums instead of int constants.

12) Avoid finalizers and properly override equals, hashCode, and toString methods.

The equals and hashCode contract must be correctly implemented to prevent hard to debug defects.

13) Write fail-fast code by validating the input parameters.Apply design by contract.

Return an empty collection or throw an exception as opposed to returning a null to avoid nullpointer exceptions which are hard to debug.

14) Security Practices:

Don’t log sensitive data.

Clearly document security related information.

Validate user inputs.

Favor immutable objects.

Security to prevent SQL injection attack.

Release resources (Streams, Connections, etc).Security to prevent denial of service attack (DoS) and resource leak issues.

Don’t let sensitive information like file paths, server names, host names, etc escape via exceptions.

Follow proper security best practices like SSL (one-way, two-way, etc), encrypting sensitive data, authentication/authorization, etc.

15) Exception Handling

Use exceptions as opposed to return codes.

Don’t ignore or suppress exceptions. Standardize the use of checked and unchecked exceptions. Throw exceptions early and catch them late.

17) Concurrency

Write thread-safe code with proper synchronization and use of immutable objects. Also, document thread-safety.

Keep synchronization section small and favor the use of the new concurrency libraries to prevent excessive synchronization.

18) Performance

Reuse objects via flyweight design pattern.

Badly constructed SQL, REGEX, etc.

Inefficient Java coding and algorithms in frequently executed methods leading to death by thousand cuts.

Interview Questions

1. Java中的原始数据类型都有哪些，它们的大小及对应的封装类是什么？
2. 谈一谈”==“与”equals()"的区别。
3. Java中的四种引用及其应用场景是什么？ 强引用: 通常我们使用new操作符创建一个对象时所返回的引用即为强引用 软引用: 若一个对象只能通过软引用到达，那么这个对象在内存不足时会被回 收，可用于图片缓存中，内存不足时系统会自动回收不再使用的Bitmap 弱引用: 若一个对象只能通过弱引用到达，那么它就会被回收（即使内存充 足），同样可用于图片缓存中，这时候只要Bitmap不再使用就会被回收 虚引用: 虚引用是Java中最“弱”的引用，通过它甚至无法获取被引用的对象，它 存在的唯一作用就是当它指向的对象回收时，它本身会被加入到引用队列中，这 样我们可以知道它指向的对象何时被销毁。
4. object中定义了哪些方法？ clone(), equals(), hashCode(), toString(), notify(), notifyAll(), wait(), finalize(), getClass()
5. hashCode的作用是什么？
6. ArrayList, LinkedList, Vector的区别是什么？ ArrayList: 内部采用数组存储元素，支持高效随机访问，支持动态调整大小 LinkedList: 内部采用链表来存储元素，支持快速插入/删除元素，但不支持高效 地随机访问 2016-06-08 absfree Android开发探索 Vector: 可以看作线程安全版的ArrayList
7. String, StringBuilder, StringBuffer的区别是什么？ String: 不可变的字符序列，若要向其中添加新字符需要创建一个新的String对象 StringBuilder: 可变字符序列，支持向其中添加新字符（无需创建新对象） StringBuffer: 可以看作线程安全版的StringBuilder
8. Map, Set, List, Queue、Stack的特点及用法。 Map: Java中存储键值对的数据类型都实现了这个接口，表示“映射表”。 支持的两个核心操作是get(Object key)以及put(K key, V value)，分别用来获取键 对应的值以及向映射表中插入键值对。 Set: 实现了这个接口的集合类型中不允许存在重复的元素，代表数学意义上 的“集合”。它所支持的核心操作有add(E e), remove(Object o), contains(Object o)，分别用于添加元素，删除元素以及判断给定元素是否存在于集中。 List: Java中集合框架中的列表类型都实现了这个接口，表示一种有序序列。 支持get(int index), add(E e)等操作。 Queue: Java集合框架中的队列接口，代表了“先进先出”队列。支持add(E element), remove()等操作。 Stack: Java集合框架中表示堆栈的数据类型，堆栈是一种“后进先出”的数据 结构。支持push(E item), pop()等操作。 更详细的说明请参考官方文档，对相关数据结构不太熟悉的同学可以参考《算法导 论》或其他相关书籍。
9. HashMap和HashTable的区别。 HashTable是线程安全的，而HashMap不是 HashMap中允许存在null键和null值，而HashTable中不允许 更加详细的信息请点击“阅读原文”
10. HashMap的实现原理 简单的说，HashMap的底层实现是“基于拉链法的散列表”。更加详细的分析请点 击“阅读原文”。
11. ConcurrentHashMap的实现原理 ConcurrentHashMap是支持并发读写的HashMap，它

的特点是读取数据时无需加 锁，写数据时可以保证加锁粒度尽可能的小。由于其内部采用“分段存储”，只需对要 进行写操作的数据所在的“段”进行加锁。关于ConcurrentHashMap底层实现的更加 详细分析请点击“阅读原文”。

12. TreeMap, LinkedHashMap, HashMap的区别是什么？ HashMap的底层实现是散列表，因此它内部存储的元素是无序的； TreeMap的底层实现是红黑树，所以它内部的元素的有序的。排序的依据是自然 序或者是创建TreeMap时所提供的比较器（Comparator）对象。 LinkedHashMap能够记住插入元素的顺序。
13. Collection与Collections的区别是什么？ Collection是Java集合框架中最基本的接口；Collections是Java集合框架提供的一个工具类，包含了大量用于操作或返回集合的静态方法。
14. 对于“try-catch-finally”，若try语句块中包含“return”语句，finally语句块会执行 吗？ 答案是会执行。只有两种情况finally块中的语句不会被执行： 调用了System.exit()方法； JVM“崩溃”了。
15. Java中的异常层次结构 更加详细的说明请点击“阅读原文”。
16. Java面向对象的三个特征与含义
17. Override, Overload的含义与区别 Override表示“重写”，是子类对父类中同一方法的重新定义 Overload表示“重载”，也就是定义一个与已定义方法名称相同但签名不同的新方 法
18. 接口与抽象类的区别 接口是一种约定，实现接口的类要遵循这个约定；抽象类本质上是一个类，使用抽 象类的代价要比接口大。接口与抽象类的对比如下： 抽象类中可以包含属性，方法（包含抽象方法与有着具体实现的方法），常量； 接口只能包含常量和方法声明。 抽象类中的方法和成员变量可以定义可见性（比如public、private等）；而接口中的方法只能为public（缺省为public）。 一个子类只能有一个父类（具体类或抽象类）；而一个接口可以继承一个多个接 口，一个类也可以实现多个接口。 子类中实现父类中的抽象方法时，可见性可以大于等于父类中的；而接口实现类 中的接口 方法的可见性只能与接口中相同（public）。
19. 静态内部类与非静态内部类的区别 静态内部类不会持有外围类的引用，而非静态内部类会隐式持有外围类的一个引 用。关于内部类的详细介绍请点击“阅读原文”。
20. Java中多态的实现原理 所谓多态，指的就是父类引用指向子类对象，调用方法时会调用子类的实现而不是 父类的实现。多态的实现的关键在于“动态绑定”。详细介绍请点击“阅读原文”。
21. 简述Java中创建新线程的两种方法 继承Thread类（假设为MyThread），并重写run()方法，然后new一个MyThread 对象并对其调用start()即可启动新线程。 实现Runnable接口（假设实现类为MyRunnable），而后将MyRunnable对象作为 参数传入Thread构造器，在得到的Thread对象上调用start()方法即可。
22. 简述Java中进行线程同步的方法 volatile: Java Memory Model保证了对同一个volatile变量的写happens before对 它的读； synchronized: 可以来对一个代码块或是对一个方法上锁，被“锁住”的地方称为 临界区，进入临界区的线程会获取对象的monitor，这样其他尝试进入临界区的 线程会因无法获取monitor而被阻塞。由于等待另一个线程释放monitor而被阻塞 的线程无法被中断。 ReentrantLock: 尝试获取锁的线程可以被中断并可以设置超时参数。 更加详细的介绍请点击“阅读原文”。

23. 简述Java中具有哪几种粒度的锁 Java中可以对类、对象、方法或是代码块上锁。更加详细的介绍请点击“阅读原 文”。
24. 给出“生产者-消费者”问题的一种解决方案
25. ThreadLocal的设计理念与作用 ThreadLocal的作用是提供线程内的局部变量，在多线程环境下访问时能保证各个线 程内的ThreadLocal变量各自独立。也就是说，每个线程的ThreadLocal变量是自己 专用的，其他线程是访问不到的。ThreadLocal最常用于以下这个场景：多线程环境 下存在对非线程安全对象的并发访问，而且该对象不需要在线程间共享，但是我们 不想加锁，这时候可以使用ThreadLocal来使得每个线程都持有一个该对象的副本。
26. concurrent包的整体架构
27. ArrayBlockingQueue, CountDownLatch类的作用 CountDownLatch: 允许线程集等待直到计数器为0。适用场景: 当一个或多个线 程需要等待指定数目的事件发生后再继续执行。 ArrayBlockingQueue: 一个基于数组实现的阻塞队列，它在构造时需要指定容 量。当试图向满队列中添加元素或者从空队列中移除元素时，当前线程会被阻 塞。通过阻塞队列，我们可以按以下模式来工作：工作者线程可以周期性的将中 间结果放入阻塞队列中，其它线程可以取出中间结果并进行进一步操作。若工作 者线程的执行比较慢（还没来得及向队列中插入元素），其他从队列中取元素的 线程会等待它（试图从空队列中取元素从而阻塞）；若工作者线程执行较快（试 图向满队列中插入元素），则它会等待其它线程取出元素再继续执行。
28. wait()，sleep()的区别 wait(): Object类中定义的实例方法。在指定对象上调用wait方法会让当前线程进 入等待状态（前提是当前线程持有该对象的monitor），此时当前线程会释放相 应对象的monitor，这样一来其它线程便有机会获取这个对象的monitor了。当其 它线程获取了这个对象的monitor并进行了所需操作时，便可以调用notify方法唤 醒之前进入等待状态的线程。 sleep(): Thread类中的静态方法，作用是让当前线程进入休眠状态，以便让其他 线程有机会执行。进入休眠状态的线程不会释放它所持有的锁。
29. 线程池的用法与优势 优势: 实现对线程的复用，避免了反复创建及销毁线程的开销；使用线程池统一 管理线程可以减少并发线程的数目，而线程数过多往往会在线程上下文切换上以 及线程同步上浪费过多时间。 用法: 我们可以调用ThreadPoolExecutor的某个构造方法来自己创建一个线程 池。但通常情况下我们可以使用Executors类提供给我们的静态工厂方法来更方 便的创建一个线程池对象。创建了线程池对象后，我们就可以调用submit方法提 交任务到线程池中去执行了；线程池使用完毕后我们要记得调用shutdown方法 来关闭它。 关于线程池的详细介绍以及实现原理分析请点击“阅读原文”。
30. for-each与常规for循环的效率对比 关于这个问题我们直接看《Effective Java》给我们做的解答： for-each能够让代码更加清晰，并且减少了出错的机会。下面的惯用代码适用于集 合与数组类型： for (Element e : elements) { doSomething(e); } 使用for-each循环与常规的for循环相比，并不存在性能损失，即使对数组进行迭代 也是如此。实际上，在有些场合下它还能带来微小的性能提升，因为它只计算一次 数组索引的上限。
31. 简述Java IO与NIO的区别 Java IO是面向流的，这意味着我们需要每次从流中读取一个或多个字节，直到 读取完所有字节；NIO是面向缓冲的，也就是说会把数据读取到一个缓冲区中， 然后对缓冲区中的数据进行相应处理。 Java IO是阻塞IO，而NIO是非阻塞

IO。 Java NIO中存在一个称为选择器（selector）的东西，它允许你把多个通道（channel）注册到一个选择器上，然后使用一个线程来监视这些通道：若这些 通道里有某个准备好可以开始进行读或写操作了，则开始对相应的通道进行读 写。而在等待某通道变为可读/写期间，请求对通道进行读写操作的线程可以去 干别的事情。

32. 反射的作用与原理 反射的作用概括地说是运行时获取类的各种定义信息，比如定义了哪些属性与方 法。原理是通过类的class对象来获取它的各种信息。
33. Java中的泛型机制 关于泛型机制的详细介绍请直接戳“阅读原文”
34. Java 1.7与1.8的新特性
35. 常见设计模式 所谓“设计模式”，不过是面向对象编程中一些常用的软件设计手法，并且经过实践 的检验，这些设计手法在各自的场景下能解决一些需求，因此它们就成为了如今广 为流传的”设计模式“。也就是说，正式因为在某些场景下产生了一些棘手的问题，才 催生了相应的设计模式。明确了这一点，我们在学习某种设计模式时要充分理解它 产生的背景以及它所解决的主要矛盾是什么。 常用的设计模式可以分为以下三大类： 创建型模式: 包括工厂模式（又可进一步分为简单工厂模式、工厂方法模式、抽 象工厂模式）、建造者模式、单例模式。 结构型模式: 包括适配器模式、桥接模式、装饰模式、外观模式、享元模式、代 理模式。 行为型模式: 包括命令模式、中介者模式、观察者模式、状态模式、策略模式。 关于每个模式具体的介绍请点击“阅读原文”。
36. JNI的基本用法
37. 动态代理的定义、应用场景及原理
38. 注解的基本概念与使用 注解可以看作是“增强版的注释”，它可以向编译器、虚拟机说明一些事情。 注解是描述Java代码的代码，它能够被编译器解析，注解处理工具在运行时也能 Read more 够解析注解。注解本身是“被动”的信息，只有主动解析它才有意义。 除了向编译器/虚拟机传递信息，我们也可以使用注解来生成一些“模板化”的代 码。

Best Practices and Guidance for Unit Testing

1. Unit Testing General Rules

```
1. Should not depend on any code or source outside the unit tested. Where there are de
pendencies they should be replaced by false objects, Mocks, and stubs.
2. Should not require manual intervention.
3. Should not manipulate real data on any environment.
4. Test cases should be automatically be re-runnable in the exact same way.
5. Do not skip unit tests:
    * Do not use JUnit's @Ignore annotation.
    * Do not use Maven's maven.test.skip property.
6. Test coverage should be at least 80%.
```

2 Naming Convention

2.1 Maven standard directory layout:

Directory	Description
src/main/java	Application/Library sources
src/main/resources	Application/Library resources
src/test/java	Test sources
src/test/resources	Test resources
src/it/java	Integration test sources
src/it/resources	Integration test resources

2.2 Test case names

```
// source class
public class org.finra.app.Foo {
    public void createOrder() {}
}

// Test class
public class org.finra.app.FooTest{
    @Test
    public void testCreateOrder() {}
}
```

3. Implementation Practices

3.1 Do not initialize in a unit test class constructor

Use of @Before, @After – @Before and @After are annotations that can be use to tag a method if you want it to be called upon initialization and destruction of test object respectively.

```
@Before
public void setData(){
    this.totalNumberOfApplicants = 9;
    listOfValidStrings.add("object_1");
    listOfValidStrings.add("object_2");
    listOfValidStrings.add("object_3");
}
@After // tearDown()
public void after() throws Exception {
    dummyAccount = null;
    assertNull(dummyAccount);
}
```

3.2 Use the most appropriate assertion methods

JUnit has many assertion methods:

- assertEquals
- assertTrue
- assertFalse
- assertNull
- assertNotNull
- assertArrayEquals
- assertSame

```
// Good
assertTrue(classUnderTest.methodUnderTest())
// Inappropriate
assertEquals(true, classUnderTest.methodUnderTest()).

// Good
assertEquals(expectedReturnValue, classUnderTest.methodUnderTest())
// Inappropriate
assertTrue(classUnderTest.methodUnderTest().equals(expectedReturnValue)).

// Good
assertEquals(expectedCollection, classUnderTest.getCollection())
// Rather than asserting on the collection's size and each of the collection's members
.
```

3.3 Do not use System.out.println() to verify test case

3.4 Do not unit-test configuration settings

You may test it in integration testing.

3.5 Mock out all external services and state

- Benefits: reduced code dependency and faster tests execution.
- Mocks are prerequisites for fast execution of tests and ability to concentrate on a single unit of functionality.
- Otherwise, behavior in those external services overlaps multiple tests, and state data means that different unit tests can influence each other’s outcome.

4. JUnit Quick Guide

You may learn more at https://github.com/junit-team/junit4/wiki

4.1 Assertions

```
    @Test
    public void testAssertArrayEquals() {

        byte[] expected = "trial".getBytes();
        byte[] actual = "trial".getBytes();
        org.junit.Assert.assertArrayEquals("failure - byte arrays not same", expected,
actual);
    }
```

```
    @Test
    public void testAssertEquals() {
        assertEquals("failure - strings are not equal", "text", "text");
    }

    @Test
    public void testAssertFalse() {
        boolean test = false;
        assertFalse("failure - should be false", test);
    }

    @Test
    public void testAssertNotNull() {
        assertNotNull("should not be null", new Object());
    }

    @Test
    public void testAssertNotSame() {
        assertNotSame("should not be same Object", new Object(), new Object());
    }

    @Test
    public void testAssertNull() {
        assertNull("should be null", null);
    }

    @Test
    public void testAssertSame() {
        Integer aNumber = Integer.valueOf(768);
        assertSame("should be same", aNumber, aNumber);
    }

    // JUnit Matchers assertThat
    @Test
    public void testAssertThat(){
        int x = 5;
        String responseString = "color";
        assertThat(x, is(5));
        assertThat(x, is(not(4)));
        assertTrue(responseString.contains("color") || responseString.contains("colour
"));
        assertThat(responseString, anyOf(containsString("color"), containsString("colo
ur")));

    }
    @Test
    public void testAssertThatBothContainsString() {
        assertThat("albumen", both(containsString("a")).and(containsString("b")));
    }

    @Test
    public void testAssertThathasItemsContainsString() {
```

```
        assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));
    }

    @Test
    public void testAssertThatEveryItemContainsString() {
        assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }), everyItem(cont
ainsString("n")));
    }
```

4.2 Parameterized Tests

When running a parameterized test class, instances are created for the cross-product of the test methods and the test data elements.

```
// Source class
public class DomainUtils {
    private static Pattern pDomainName;
    private static final String DOMAIN_NAME_PATTERN = "^((?!-)[A-Za-z0-9-]{1,63}(?<!-)
\\.)+[A-Za-z]{2,6}$";
    static {
        pDomainName = Pattern.compile(DOMAIN_NAME_PATTERN);
    }

    // Check if it is a valid domain name
    public static boolean isValidDomainName(String domainName) {
        return pDomainName.matcher(domainName).find();
    }
}
```

```
// Test class
@RunWith(value = Parameterized.class)
public class DomainUtilsTest {
    private String domain;
    private boolean expected;

    public DomainUtilsTest(String domain, boolean expected) {
        this.domain = domain;
        this.expected = expected;
    }

    @Parameters
    public static Iterable<Object[]> data() {
        return Arrays.asList(new Object[][] {
                        { "google.com", true },
                        { "finra.org", true },
                        { "-finra.org", false },
                        { "finra-.com", false },
                        { "3423kjk", false },
                        { "fi#$kdo.com", false }
                }
        );
    }

    @Test
    public void test_validDomains() {
        assertEquals(expected,DomainUtils.isValidDomainName(domain));
    }
}
```

5. Mockito Quick Guide

The Mockito library enables mock creation, verification and stubbing. Learn more at http://static.javadoc.io/org.mockito/mockito-core/2.2.9/org/mockito/Mockito.html#2

5.1 Stubbing

```
//You can mock concrete classes, not just interfaces
 LinkedList mockedList = mock(LinkedList.class);

 //stubbing
 when(mockedList.get(0)).thenReturn("first");
 when(mockedList.get(1)).thenThrow(new RuntimeException());

 //following prints "first"
 System.out.println(mockedList.get(0));

 //following throws runtime exception
 System.out.println(mockedList.get(1));

 //following prints "null" because get(999) was not stubbed
 System.out.println(mockedList.get(999));

 //Although it is possible to verify a stubbed invocation, usually it's just redundant
 //If your code cares what get(0) returns, then something else breaks (often even befo
re verify() gets executed).
 //If your code doesn't care what get(0) returns, then it should not be stubbed. Not c
onvinced? See here.
 verify(mockedList).get(0);
```

5.2 Verification

```
LinkedList mockedList = mock(LinkedList.class);
//using mock
 mockedList.add("once");

 mockedList.add("twice");
 mockedList.add("twice");

 mockedList.add("three times");
 mockedList.add("three times");
 mockedList.add("three times");

 //following two verifications work exactly the same - times(1) is used by default
 verify(mockedList).add("once");
 verify(mockedList, times(1)).add("once");

 //exact number of invocations verification
 verify(mockedList, times(2)).add("twice");
 verify(mockedList, times(3)).add("three times");

 //verification using never(). never() is an alias to times(0)
 verify(mockedList, never()).add("never happened");

 //verification using atLeast()/atMost()
 verify(mockedList, atLeastOnce()).add("three times");
 verify(mockedList, atLeast(2)).add("five times");
 verify(mockedList, atMost(5)).add("three times");
```

5.3 Mock dependencies

```
//Source code
public class Foo {
    private Bar bar;
    public One(Bar bar) {
        this.bar = bar;
    }
    public void work(){
        bar.doSomething();
    }
}

public class Bar {
    public void doSomething(){
        System.out.println("Bar is doing something...");
    }
}
```

```
//Test code
public class FooTest {
    @InjectMocks
    private Foo foo;

    @Mock
    private Bar bar;

    @Before
    public void setup() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void testWork() {
        foo.work();
        Mockito.verify(bar).doSomething();
    }
}
```

Integration Testing

List

No wildcard imports.
Overloads appear sequentially.
Braces are used even when the body is empty or contains a single statement.
Two space indentation.
Column limit can be 80 or 100 characters.
No C-style array declarations.
Default statements required in switch statements.
Modifiers appear in the order recommended by the Java Language Specification.
Constants use CONSTANT_CASE. Note that every constant is a static final field, but not all static final fields are constants.

代码审查清单

常规项

> 代码能够工作么？它有没有实现预期的功能，逻辑是否正确等。
> 所有的代码是否简单易懂？
> 代码符合你所遵循的编程规范么？这通常包括大括号的位置，变量名和函数名，行的长度，缩进，格式和注释。
> 是否存在多余的或是重复的代码？
> 代码是否尽可能的模块化了？
> 是否有可以被替换的全局变量？
> 是否有被注释掉的代码？
> 循环是否设置了长度和正确的终止条件？
> 是否有可以被库函数替代的代码？
> 是否有可以删除的日志或调试代码？

安全

> 所有的数据输入是否都进行了检查（检测正确的类型，长度，格式和范围）并且进行了编码？
> 在哪里使用了第三方工具，返回的错误是否被捕获？
> 输出的值是否进行了检查并且编码？
> 无效的参数值是否能够处理？

文档

> 是否有注释，并且描述了代码的意图？
> 所有的函数都有注释吗？
> 对非常规行为和边界情况处理是否有描述？
> 第三方库的使用和函数是否有文档？
> 数据结构和计量单位是否进行了解释？
> 是否有未完成的代码？如果是的话，是不是应该移除，或者用合适的标记进行标记比如‘TODO’？

测试

> 代码是否可以测试？比如，不要添加太多的或是隐藏的依赖关系，不能够初始化对象，测试框架可以使用方法等。
> 是否存在测试，它们是否可以被理解？比如，至少达到你满意的代码覆盖(code coverage)。
> 单元测试是否眞正的测试了代码是否可以完成预期的功能？
> 是否检查了数组的“越界“错误？
> 是否有可以被已经存在的API所替代的测试代码？

你同样需要把特定语言中有可能引起错误的问题添加到清单中。

这个清单故意没有详尽的列出所有可能会发生的错误。你不希望你的清单是这样的，太长了以至于从来没人会去用它。仅仅包含常见的问题会比较好。

优化你的清单

把使用清单作为你的起点，针对特定的使用案例，你需要对其进行优化。一个比较棒的方式就是让你的团队记录下那些在代码审查过程中临时发现的问题，有了这些数据，你就能够确定你的团队常犯的错误，然后你就可以量身定制一个审查清单。确保你删除了那些没有出现过的错误。（你也可以保留那些出现概率很小，但是非常关键的项目，比如安全相关的问题）。

得到认可并且保持更新

基本规则是，清单上的任何条目都必须明确，而且，如果可能的话，对于一些条目你可以对其进行二元判定。这样可以防止判断的不一致。和你的团队分享这份清单并且让他们认同你清单的内容是个好主意。同样的，要定期检查你的清单，以确保各条目仍然是有意义的。

有了一个好的清单，可以提高你在代码审查过程中发现的缺陷个数。这可以帮助你提高代码标准，避免质量参差不齐的代码审查。

How does HashMap work in Java

1. Internal Storage

1.1 Internal Structure

The JAVA HashMap class implements the interface Map. The main methods of this interface are:

```
* V put(K key, V value)
* V get(Object key)
* V remove(Object key)
* Boolean containsKey(Object key)
```

HashMaps use an inner class to store data: the `Entry<K, V>`. This entry is a simple key-value pair with two extra data:

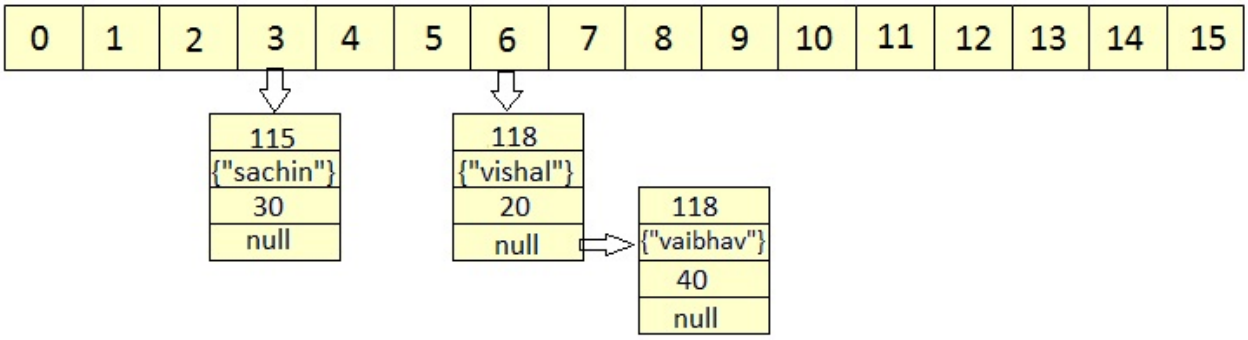
```
* A reference to another Entry so that a HashMap can store entries like singly linked
lists
* A hash value that represents the hash value of the key. This hash value is stored to
 avoid the computation of the hash every time the HashMap needs it.
```

Here is a part of the Entry implementation in JAVA 7:

```
static class Entry<K,V> implements Map.Entry<K,V> {
        final K key;
        V value;
        Entry<K,V> next;
        int hash;
…
}
```

1.2 How to CRUD data

A HashMap stores data into multiple singly linked lists of entries. All the lists are registered in an array of Entry (`Entry<K,V>[] array`) and the default capacity of this inner array is `16` .



All the keys with the same hash value are put in the same linked list (bucket). Keys with different hash values can end-up in the same bucket.

When a user calls `put(K key, V value)` or `get(Object key)` , the function computes the index of the bucket in which the Entry should be. Then, the function iterates through the list to look for the Entry that has the same key (using the equals() function of the key).

In the case of the `get()` , the function returns the value associated with the entry (if the entry exists).

In the case of the `put(K key, V value)` , if the entry exists the function replaces it with the new value otherwise it creates a new entry (from the key and value in arguments) at the head of the singly linked list.

1.3 How to calculate hashCode

This index of the bucket (linked list) is generated in 3 steps by the map:

- It first gets the hashcode of the key.
- It rehashes the hashcode to prevent against a bad hashing function from the key that would put all data in the same index (bucket) of the inner array
- It takes the rehashed hash hashcode and bit-masks it with the length (minus 1) of the array. This operation assures that the index can’t be greater than the size of the array. You can see it as a very computationally optimized modulo function.

Here is the JAVA 7 and 8 source code that deals with the index:

```
// JAVA 7: the "rehash" function
static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

```
// JAVA 8: the "rehash" function
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }
// the function that returns the index from the rehashed hash
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

In order to work efficiently, the size of the inner array needs to be a power of `2`, let's see why.

Imagine the array size is `17`, the mask value is going to be `16 (size -1)`. The binary representation of `16` is `0…010000`, so for any hash value H the index generated with the bitwise formula "`H AND 16`" is going to be either `16` or `0`. This means that the array of size `17` will only be used for 2 buckets: the one at index 0 and the one at index 16, not very efficient…

But, if you now take a size that is a power of 2 like 16, the bitwise index formula is "H AND 15". The binary representation of 15 is 0…001111 so the index formula can output values from 0 to 15 and the array of size 16 is fully used. For example:

- `if H = 952 , its binary representation is 0..01110111000, the associated index is 0…01000 = 8`
- `if H = 1576 its binary representation is 0..011000101000, the associated index is 0…01000 = 8`
- `if H = 12356146, its binary representation is 0..0101111001000101000110010, the associated index is 0…00010 = 2`
- `if H = 59843, its binary representation is 0..01110100111000011, the associated index is 0…00011 = 3`

This is why the array size is a power of two. This mechanism is transparent for the developer: if he chooses a HashMap with a size of 37, the Map will automatically choose the next power of 2 after 37 (64) for the size of its inner array.

2. Auto resizing

After getting the index, the function (`get, put or remove`) visits/iterates the associated linked list to see if there is an existing Entry for the given key. Without modification, this mechanism could lead to performance issues because the function needs to iterate through the entire list to see if the entry exists. Imagine that the size of the inner array is the default value (16) and you need to store 2 millions values. In the best case scenario, each linked list will have a size of `125 000` entries (2/16 millions). So, each `get(), remove() and put()` will lead to `125 000` iterations/operations. To avoid this case, the HashMap has the ability to increase its inner array in order to keep very short linked lists.

When you create a HashMap, you can specify an initial size and a loadFactor with the following constructor:

```
public HashMap(int initialCapacity, float loadFactor)
```

If you don't specify arguments, the default `initialCapacity` is `16` and the default `loadFactor` is `0.75` . The `initialCapacity` represents to the size of the inner array of linked lists.

Each time you add a new key/value in your Map with put(…), the function checks if it needs to increase the capacity of the inner array. In order to do that, the map stores 2 data:

- The size of the map: it represents the number of entries in the HashMap. This value is updated each time an Entry is added or removed.
- A threshold: it's equal to (capacity of the inner array) `* loadFactor` and it is refreshed after each resize of the inner array

Before adding the new Entry, put(…) checks if `size > threshold` and if it the case it recreates a new array with a doubled size. Since the size of the new array has changed, the indexing function (which returns the bitwise operation " `hash(key) AND (sizeOfArray-1)` ") changes. So, the resizing of the array creates twice more buckets (i.e. linked lists) and redistributes all the existing entries into the buckets (the old ones and the newly created).

This aim of this resize operation is to decrease the size of the linked lists so that the time cost of `put(), remove() and get()` methods stays low. All entries whose keys have the same hash will stay in the same bucket after the resizing. But, 2 entries with different hash keys that were in the same bucket before might not be in the same bucket after the transformation.

Note: the HashMap only increases the size of the inner array, it doesn't provide a way to decrease it.

3. Thread Safety

If you already know HashMaps, you know that is not threads safe, but why? For example imagine that you have a Writer thread that puts only new data into the Map and a Reader thread that reads data from the Map, why shouldn't it work?

Because during the auto-resizing mechanism, if a thread tries to put or get an object, the map might use the old index value and won't find the new bucket in which the entry is.

The worst case scenario is when 2 threads put a data at the same time and the 2 put() calls resize the Map at the same time. Since both threads modify the linked lists at the same time, the Map might end up with an inner-loop in one of its linked lists. If you tries to get a data in the list with an inner loop, the get() will never end.

The HashTable implementation is a thread safe implementation that prevents from this situation. But, since all the CRUD methods are synchronized this implementation is very slow. For example, if thread 1 calls get(key1), thread 2 calls get(key2) and thread 3 calls get(key3), only one thread at a time will be able to get its value whereas the 3 of them could access the data at the same time.

A smarter implementation of a thread safe HashMap exists since JAVA 5: the `ConcurrentHashMap`. **Only the buckets are synchronized** so multiples threads can `get(), remove() or put()` data at the same time if it doesn't imply accessing the same bucket or resizing the inner array. It's better to use this implementation in a multithreaded application.

4. Key immutability

Why `String` and `Integer` are a good implementation of keys for HashMap? Mostly because they are immutable! If you choose to create your own Key class and don't make it immutable, you might lose data inside the HashMap.

Look at the following use case:

- You have a key that has an inner value "1"
- You put an object in the HashMap with this key
- The HashMap generates a hash from the hashcode of the Key (so from "1")
- The Map stores this hash in the newly created Entry
- You modify the inner value of the key to "2"
- The hash value of the key is modified but the HashMap doesn't know it (because the old hash value is stored)
- You try to get your object with your modified key
- The map computes the new hash of your key (so from "2") to find in which linked list (bucket) the entry is

- Case 1: Since you modified your key, the map tries to find the entry in the wrong bucket and doesn't find it
- case 2: Luckily, the modified key generates the same bucket as the old key. The map then iterates through the linked list to find the entry with the same key. But to find the key, the map first compares the hash values and then calls the equals() comparison. Since your modified key doesn't have the same hash as the old hash value (stored in the entry), the map won't find the entry in the linked-list.

Here is a concrete example in Java. I put 2 key-value pairs in my Map, I modify the first key and then try to get the 2 values. Only the second value is returned from the map, the first value is "lost" in the HashMap:

```
public class MutableKeyTest {
    public static void main(String[] args) {
        class MyKey {
            Integer i;
            public void setI(Integer i) { this.i = i; }
            public MyKey(Integer i) { this.i = i;}

            @Override
            public int hashCode() {return i;}
            @Override
            public boolean equals(Object obj) {
                if (obj instanceof MyKey) {
                    return i.equals(((MyKey) obj).i);
                } else
                    return false;
            }
        }

        Map<MyKey, String> myMap = new HashMap<>();
        MyKey key1 = new MyKey(1);
        MyKey key2 = new MyKey(2);
        myMap.put(key1, "test " + 1);
        myMap.put(key2, "test " + 2);

        // modifying key1
        key1.setI(3);

        String test1 = myMap.get(key1);
        String test2 = myMap.get(key2);

        System.out.println("test1= " + test1 + " test2=" + test2);
    }
}
```

The output is: " `test1= null test2=test 2` ". As expected, the Map wasn't able to retrieve the string 1 with the modified key 1.

5. JAVA 8 improvements

The inner representation of the HashMap has changed a lot in JAVA 8. Indeed, the implementation in JAVA 7 takes 1k lines of code whereas the implementation in JAVA 8 takes 2k lines. Most of what I've said previously is true except the linked lists of entries. In JAVA8, you still have an array but it now stores Nodes that contains the exact same information as Entries and therefore are also linked lists:

Here is a part of the Node implementation in JAVA 8:

```
    static class Node<K,V> implements Map.Entry<K,V> {
         final int hash;
         final K key;
         V value;
         Node<K,V> next;
         ...
    }
```

So what's the big difference with JAVA 7? Well, Nodes can be extended to TreeNodes. A TreeNode is a red-black tree structure that stores really more information so that it can add, delete or get an element in `O(log(n))`.

FYI, here is the exhaustive list of the data stored inside a TreeNode

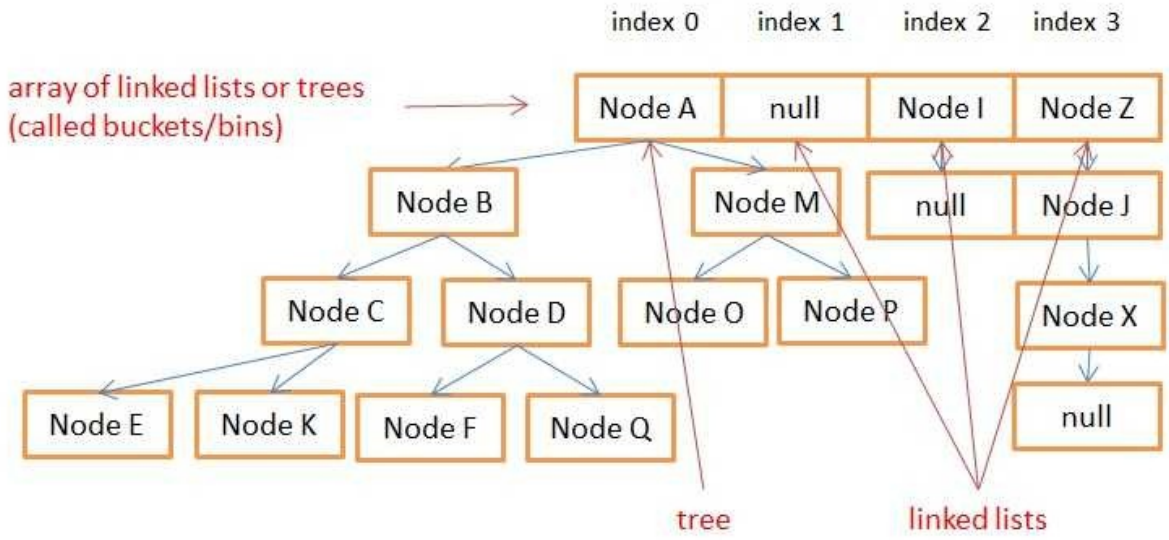
```
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
    final int hash; // inherited from Node<K,V>
    final K key; // inherited from Node<K,V>
    V value; // inherited from Node<K,V>
    Node<K,V> next; // inherited from Node<K,V>
    Entry<K,V> before, after;// inherited from LinkedHashMap.Entry<K,V>
    TreeNode<K,V> parent;
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev;
    boolean red;
}
```

Red black trees are self-balancing binary search trees. Their inner mechanisms ensure that their length is always in `log(n)` despite new adds or removes of nodes. **The main advantage** to use those trees is in a case where many data are in the same index (bucket) of the inner table, the search in a tree will cost `O(log(n))` whereas it would have cost O(n) with a linked list.

As you see, the tree takes really more space than the linked list (we'll speak about it in the next part).

By inheritance, the inner table can contain both Node (linked list) and TreeNode (red-black tree). Oracle decided to use both data structures with the following rules:

- If for a given index (bucket) in the inner table there are more than 8 nodes, the linked list is transformed into a red black tree
- If for a given index (bucket) in the inner table there are less than 6 nodes, the tree is transformed into a linked list



This picture shows an inner array of a JAVA 8 HashMap with both trees (at bucket 0) and linked lists (at bucket 1,2 and 3). Bucket 0 is a Tree because it has more than 8 nodes.

6. Memory overhead

6.1 JAVA 7

The use of a HashMap comes at a cost in terms of memory. In JAVA 7, a HashMap wraps key-value pairs in Entries. An entry has:

- a reference to a next entry
- a precomputed hash (integer)
- a reference to the key
- a reference to the value

Moreover, a JAVA 7 HashMap uses an inner array of Entry. Assuming a JAVA 7 HashMap contains N elements and its inner array has a capacity CAPACITY, the extra memory cost is approximately: `sizeOf(integer)* N + sizeOf(reference)* (3*N+C)`

Where:

- the size of an integer depends equals `4` bytes
- the size of a reference depends on the JVM/OS/Processor but is often `4` bytes. Which

means that the overhead is often `16 * N + 4 * CAPACITY` bytes

Reminder: after an auto-resizing of the Map, the CAPACITY of the inner array equals the next power of two after N.

Note: Since JAVA 7, the HashMap class has a **lazy init**. That means that even if you allocate a HashMap, the inner array of entry (that costs 4 * CAPACITY bytes) won't be allocated in memory until the first use of the put() method.

6.2 JAVA 8

With the JAVA 8 implementation, it becomes a little bit complicated to get the memory usage because a Node can contain the same data as an Entry or the same data plus 6 references and a Boolean (if it's a TreeNode).

If the all the nodes are only Nodes, the memory consumption of the JAVA 8 HashMap is the same as the JAVA 7 HashMap.

If the all the nodes are TreeNodes, the memory consumption of a JAVA 8 HashMap becomes: `N * sizeOf(integer) + N * sizeOf(boolean) + sizeOf(reference)* (9*N+CAPACITY )`

In most standards JVM, it's equal to `44 * N + 4 * CAPACITY` bytes

7. Performance issues

7.1 Skewed HashMap vs well balanced HashMap:

In the best case scenario, the `get()` and `put()` methods have a `O(1)` cost in time complexity. But, if you don't take care of the hash function of the key, you might end up with very slow `put()` and `get()` calls. The good performance of the put() and get depends on the repartition of the data into the different indexes of the inner array (the buckets). If the hash function of your key is ill-designed, you'll have a skew repartition (no matter how big the capacity of the inner array is). All the put() and get() that use the biggest linked lists of entry will be slow because they'll need to iterate the entire lists.

In the worst case scenario (if most of the data are in the same buckets), you could end up with a `O(n)` time complexity.

7.2 Resizing overhead

If you need to store a lot of data, you should create your HashMap with an initial capacity close to your expected volume.

If you don't do that, the Map will take the default size of `16` with a `factorLoad` of `0.75`. The `11` first put() will be very fast but the `12th (16*0.75)` will recreate a new inner array (with its associated linked lists/trees) with a new capacity of `32`. The `13th` to `23th` will be

fast but the `24th (32*0.75)` will recreate (again) a costly new representation that doubles the size of the inner array. The internal resizing operation will appear at the `48th, 96th,192th` , … call of put(). At low volume the full recreation of the inner array is fast but at high volume it can takes seconds to minutes. By initially setting your expected size, you can avoid these costly operations.

But there is a drawback: if you set a very high array size like `2^28` whereas you're only using `2^26` buckets in your array, you will waste a lot of memory (approximately `2^30` bytes in this case).

`HashMap` is a very powerful data structure inJava. We use it everyday and almost in all applications. There are quite a few examples which I have written before onHow to Implement Threadsafe cache, How to convertHashmap to Arraylist?

We used Hashmap in both above examples but those are pretty simple use cases of Hashmap. `HashMap is a non-synchronized` collection class.

Do you have any of below questions?

- What's the difference between ConcurrentHashMap and Collections.synchronizedMap(Map)?
- What's the difference between ConcurrentHashMap and Collections.synchronizedMap(Map) in term of performance?
- ConcurrentHashMap vs Collections.synchronizedMap()
- Popular HashMap and ConcurrentHashMap interview questions

In this tutorial we will go over all above queries and reason `why and how` we could Synchronize Hashmap?

Why?

The Map object is an associative containers that store elements, formed by a combination of a uniquely identify `key` and a mapped `value` . If you have very highly concurrent application in which you may want to modify or read key value in different threads then it's ideal to use Concurrent Hashmap. Best example isProducer Consumer which handles concurrent read/write.

So what does the thread-safe Map means? If `multiple threads` access a hash map concurrently, and at least one of the threads modifies the map structurally, it `must be synchronized externally` to avoid an inconsistent view of the contents.

How?

There are two ways we could synchronizedHashMap

1. Java Collections synchronizedMap() method
2. Use ConcurrentHashMap

```
//Hashtable
Map<String, String> normalMap = new Hashtable<String, String>();

//synchronizedMap
synchronizedHashMap = Collections.synchronizedMap(new HashMap<String, String>());

//ConcurrentHashMap
concurrentHashMap = new ConcurrentHashMap<String, String>();
```

ConcurrentHashMap

You should use ConcurrentHashMap when you need very high concurrency in your project. It is thread safe without synchronizing the whole map. Reads can happen very fast while write is done with a lock. There is no locking at the object level. The locking is at a much finer granularity at a hashmap bucket level. ConcurrentHashMap doesn't throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it. ConcurrentHashMap uses multitude of locks.

SynchronizedHashMap

Synchronization at Object level. Every read/write operation needs to acquire lock. Locking the entire collection is a performance overhead. This essentially gives access to only one thread to the entire map & blocks all the other threads. It may cause contention. SynchronizedHashMap returns Iterator, which fails-fast on concurrent modification. Now let's take a look at code

Create class CrunchifyConcurrentHashMapVsSynchronizedHashMap.java Create object for each HashTable, SynchronizedMap and CrunchifyConcurrentHashMap Add and retrieve 500k entries from Map Measure start and end time and display time in milliseconds We will use ExecutorService to run 5 threads in parallel

```
package crunchify.com.tutorials;

import java.util.Collections;
import java.util.HashMap;
import java.util.Hashtable;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

/**
 * @author Crunchify.com
 *
 */
```

```
public class CrunchifyConcurrentHashMapVsSynchronizedMap {

    public final static int THREAD_POOL_SIZE = 5;

    public static Map<String, Integer> crunchifyHashTableObject = null;
    public static Map<String, Integer> crunchifySynchronizedMapObject = null;
    public static Map<String, Integer> crunchifyConcurrentHashMapObject = null;

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

        // Test with Hashtable Object
        crunchifyHashTableObject = new Hashtable<String, Integer>();
        crunchifyPerformTest(crunchifyHashTableObject);

        // Test with synchronizedMap Object
        crunchifySynchronizedMapObject = Collections.synchronizedMap(new HashMap<Strin
g, Integer>());
        crunchifyPerformTest(crunchifySynchronizedMapObject);

        // Test with ConcurrentHashMap Object
        crunchifyConcurrentHashMapObject = new ConcurrentHashMap<String, Integer>();
        crunchifyPerformTest(crunchifyConcurrentHashMapObject);

    }

    public static void crunchifyPerformTest(final Map<String, Integer> crunchifyThread
s) throws InterruptedException {

        System.out.println("Test started for: " + crunchifyThreads.getClass());
        long averageTime = 0;
        for (int i = 0; i < 5; i++) {

            long startTime = System.nanoTime();
            ExecutorService crunchifyExServer = Executors.newFixedThreadPool(THREAD_PO
OL_SIZE);

            for (int j = 0; j < THREAD_POOL_SIZE; j++) {
                crunchifyExServer.execute(new Runnable() {
                    @SuppressWarnings("unused")
                    @Override
                    public void run() {

                        for (int i = 0; i < 500000; i++) {
                            Integer crunchifyRandomNumber = (int) Math.ceil(Math.rando
m() * 550000);

                            // Retrieve value. We are not using it anywhere
                            Integer crunchifyValue = crunchifyThreads.get(String.value
Of(crunchifyRandomNumber));

                            // Put value
                            crunchifyThreads.put(String.valueOf(crunchifyRandomNumber)
, crunchifyRandomNumber);
```

```
                        }
                    }
                });
            }

            // Make sure executor stops
            crunchifyExServer.shutdown();

            // Blocks until all tasks have completed execution after a shutdown request

            crunchifyExServer.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);

            long entTime = System.nanoTime();
            long totalTime = (entTime - startTime) / 1000000L;
            averageTime += totalTime;
            System.out.println("2500K entried added/retrieved in " + totalTime + " ms"
);
        }
        System.out.println("For " + crunchifyThreads.getClass() + " the average time i
s " + averageTime / 5 + " ms\n");
    }
}
```