Java

1. Data Types, Variables, Arrays.

Java Is a Strongly Typed Language

The Java compiler checks all expressions and parameters to ensure that the types are compatible.

The Primitive Types

Integers

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages support both signed and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of *unsigned* was used mostly to specify the behavior of the *high-order bit*, which defines the *sign* of an integer value. As you will see in Chapter 4, Java manages the meaning of the high-order bit differently, by adding a special "unsigned right shift" operator. Thus, the need for an unsigned integer type was eliminated.

The *width* of an integer type should not be thought of as the amount of storage it consumes, but rather as the *behavior* it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. The width and ranges of these integer types vary widely, as shown in this table:

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Floating-Point Types

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating-point type. Java implements the standard (IEEE–754) set of floating-point types and operators. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

Each of these floating-point types is examined next.

float

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision. For example, **float** can be useful when representing dollars and cents.

Here are some example **float** variable declarations:

```
float hightemp, lowtemp;
```

double

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, **double** is the best choice.

Characters

In Java, the data type used to store characters is **char**. A key point to understand is that Java uses *Unicode* to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. At the time of Java's creation, Unicode required 16 bits. Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,535. There are no negative **char**s. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits. But such is the price that must be paid for global portability.

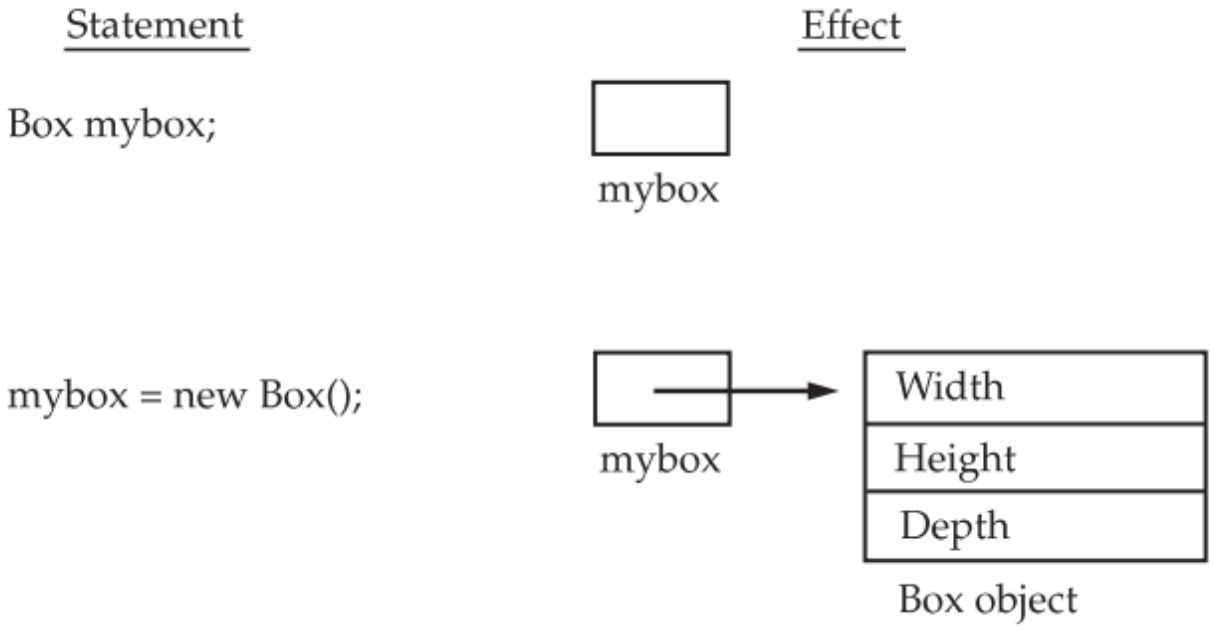
2. Classes

A Closer Look at new

As just explained, the **new** operator dynamically allocates memory for an object. In the context of an assignment, it has this general form:

class-var = new *classname* ( );

Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. For now, we will use the default constructor. Soon, you will see how to define your own constructors.



At this point, you might be wondering why you do not need to use **new** for such things as integers or characters. The answer is that Java's primitive types are not implemented as objects. Rather, they are implemented as "normal" variables. This is done in the interest of efficiency. As you will see, objects have many features and attributes that require Java to treat them differently than it treats the primitive types. By not applying the same overhead to the primitive types that applies to objects, Java can implement the primitive types more efficiently. Later, you will see object versions of the primitive types that are available for your use in those situations in which complete objects of these types are needed.

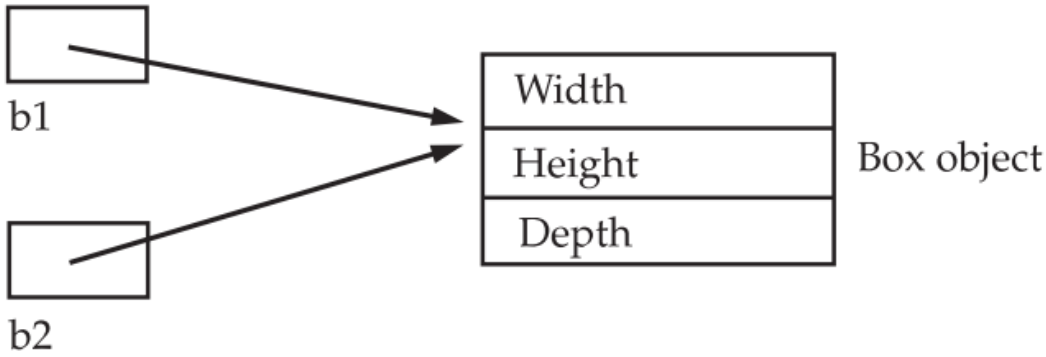
It is important to understand that **new** allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur. (You will learn how to handle exceptions in Chapter 10.) For the sample programs in this book, you won't need to worry about running out of memory, but you will need to consider this possibility in real-world programs that you write.

Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

This situation is depicted here:



Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as traditional C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no need to explicitly destroy objects. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

Some of the ways to make new instance without new:

```
public static void main(String[] args) throws Exception {
    Dummy dummy1 = (Dummy) Class.forName("reflection.Dummy")
                                .getConstructor()
                                .newInstance();
    Dummy dummy2 = (Dummy) Dummy.class.getClassLoader()
                                .loadClass("reflection.Dummy")
                                .getDeclaredConstructors()[0]
                                .newInstance();
    Dummy dummy3 = (Dummy) dummy2.clone();
}
```

A Closer Look at Argument Passing

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is *call-by-value*. This approach copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is *call-by-reference*. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, although Java uses call-by-value to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed.

When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
// Primitive types are passed by value.
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}
```

As you can see, the operations that occur inside **meth( )** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and 10.

When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects act as if they are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument. For example, consider the following program:

Local Variable Type Inference with Reference Types

As you saw in Chapter 3, beginning with JDK 10, Java supports local variable type inference. Recall that when using local variable type inference, the type of the variable is specified as **var** and the variable must be initialized. Earlier examples have shown type inference with primitive types, but it can also be used with reference types. In fact, type inference with reference types constitutes a primary use. Here is a simple example that declares a **String** variable called **myStr**:

```
var myStr = "This is a string";
```

Because a quoted string is used as an initializer, the type **String** is inferred.

As explained in Chapter 3, one of the benefits of local variable type inference is its ability to streamline code, and it is with reference types where such streamlining is most apparent.

3. Inheritance

Dynamic Method Dispatch

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Using **final** with Inheritance:

- Using final to prevent override method
- Using final to inheritance class

The Object Class

There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**. That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.

Object defines the following methods, which means that they are available in every object.

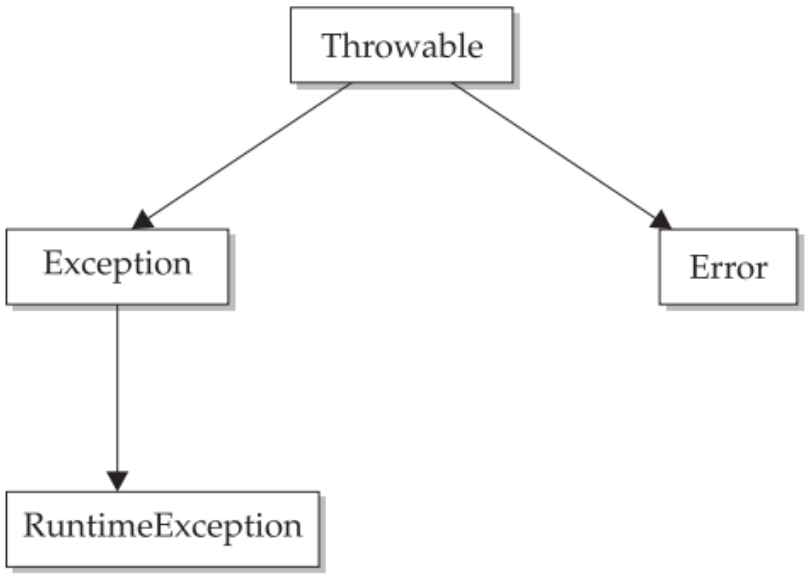
Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled. (Deprecated by JDK 9.)
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

4. Exception handling

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type **Error**, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

The top-level exception hierarchy is shown here:




```java
public class ExceptionTest {
    void thisFunctionThrowRuntime(){
        throw new RuntimeException("Test");
    }

    void thisFunctionThrowCheckException() throws Exception {
        // Only Check Exception require change method signature
        throw new Exception("Test");
    }

    void thisFunctionThrowError(){
        throw new Error("Test");
    }
}
```

Some Uncheck Exception

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalCallerException	A method cannot be legally executed by the calling code.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
LayerInstantiationException	A module layer cannot be created.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Some check Exception

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

5. Multithread programming

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

You are almost certainly acquainted with multitasking because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two. For many readers, process-based multitasking is the more familiar form. A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

Java multithreading model

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an *event loop* with *polling*. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the program. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a single-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Thread State

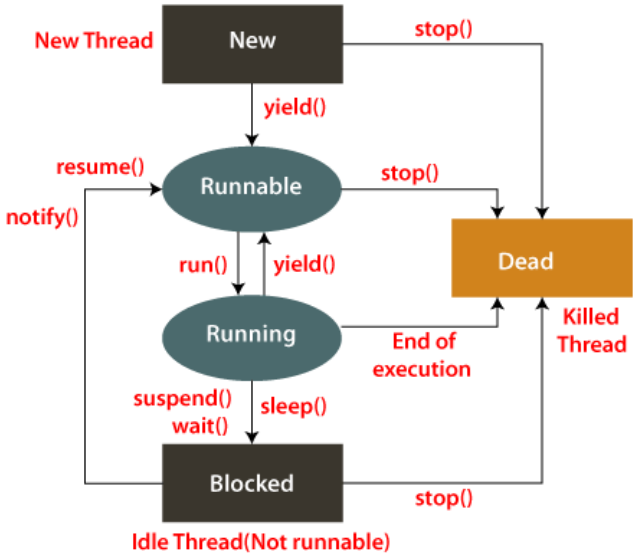


Fig: State Transition Diagram of a Thread

Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*. The rules that determine when a context switch takes place are simple:

- *A thread can voluntarily relinquish control*. This occurs when explicitly yielding, sleeping, or when blocked. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- *A thread can be preempted by a higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking*.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For some operating systems, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

Synchronization

The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

In Java, there is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

The Thread Class and the Runnable Interface

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it. To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

The **Thread** class defines several methods that help manage threads. Several of those used in this chapter are shown here:

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

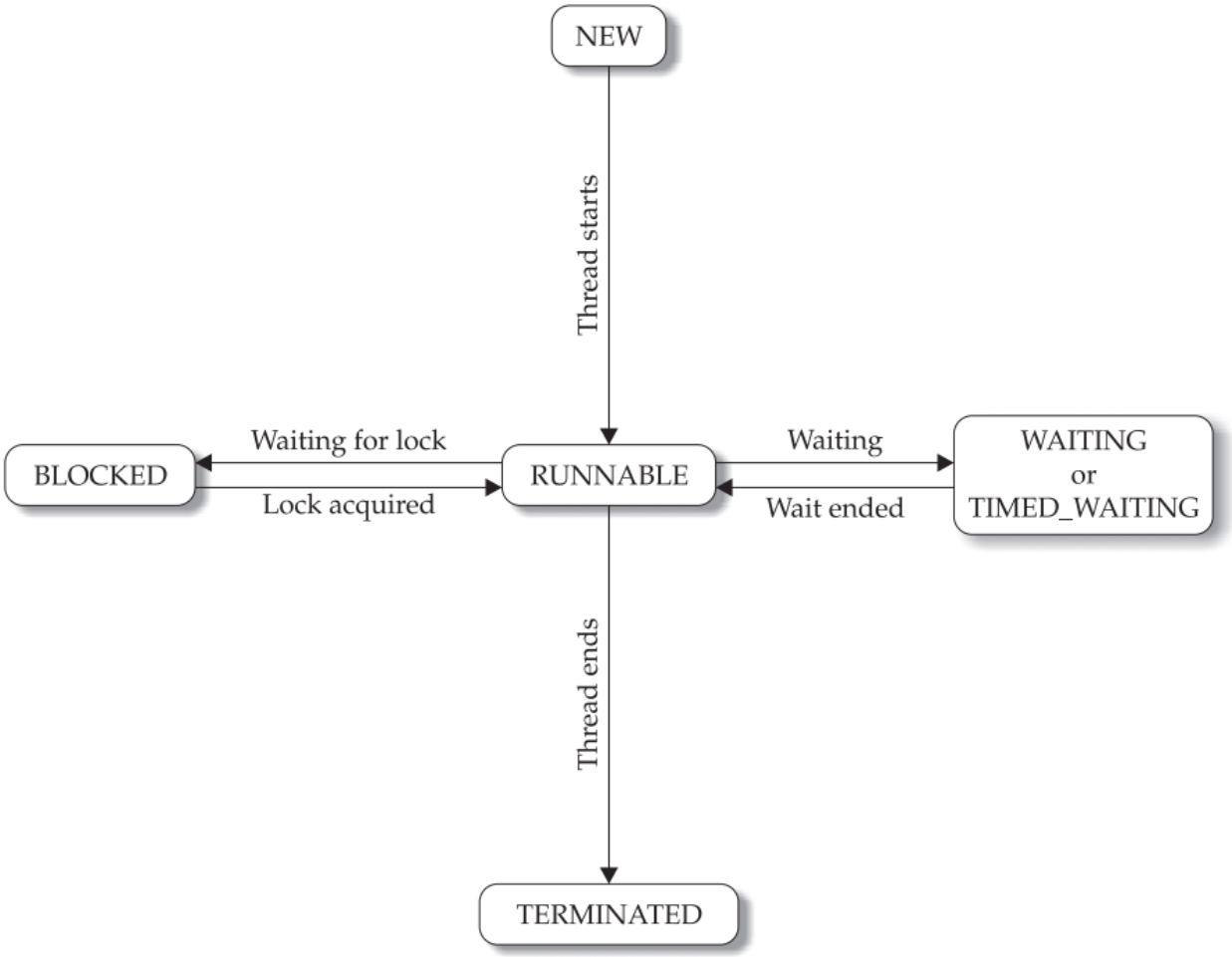
Creating Thread

```
class NewThread implements Runnable {
    Thread t;
    NewThread() {

        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
    }

    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}
```

Using isAlive( ) and join( )

- **isAlive**: use for checking thread state is running or not
- **join**: wait for thread to be exterminated

```
class Q {
    int n;
    boolean valueSet = false;

    synchronized void get() throws InterruptedException {
        while(!valueSet){
            try {
                wait();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
        }
        Thread.sleep(500);
        System.out.println("Got: " + n);
        valueSet = false;
        notify();
    }

    synchronized void put(int n) throws InterruptedException {
        while(valueSet){
            try {
                wait();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
        }
        Thread.sleep(500);
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
}
```



6. Annotation

Java provides a feature that enables you to embed supplemental information into a source file. This information, called an *annotation*, does not change the actions of a program. Thus, an annotation leaves the semantics of a program unchanged. However, this information can be used by various tools during both development and deployment. For example, an annotation might be processed by a source-code generator. The term *metadata* is also used to refer to this feature, but the term *annotation* is the most descriptive and more commonly used.

Annotation Basics

An annotation is created through a mechanism based on the **interface**. Let's begin with an example. Here is the declaration for an annotation called **MyAnno**:

```
// A simple annotation type.
@interface MyAnno {
  String str();
  int val();
}
```

First, notice the @ that precedes the keyword **interface**. This tells the compiler that an annotation type is being declared. Next, notice the two members **str( )** and **val( )**. All annotations consist solely of method declarations. However, you don't provide bodies for these methods. Instead, Java implements these methods. Moreover, the methods act much like fields, as you will see.

An annotation cannot include an **extends** clause. However, all annotation types automatically extend the **Annotation** interface. Thus, **Annotation** is a super-interface of all annotations. It is declared within the **java.lang.annotation** package. It overrides **hashCode( )**, **equals( )**, and **toString( )**, which are defined by **Object**. It also specifies **annotationType( )**, which returns a **Class** object that represents the invoking annotation.

When you apply an annotation, you give values to its members. For example, here is an example of **MyAnno** being applied to a method declaration:

```
// Annotate a method.
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth() { // ...
```

This annotation is linked with the method **myMeth( )**. Look closely at the annotation syntax. The name of the annotation, preceded by an @, is followed by a parenthesized list of member initializations. To give a member a value, that member's name is assigned a value. Therefore, in the example, the string "Annotation Example" is assigned to the **str** member of **MyAnno**. Notice that no parentheses follow **str** in this assignment. When an annotation member is given a value, only its name is used. Thus, annotation members look like fields in this context.

Specifying a Retention Policy

Before exploring annotations further, it is necessary to discuss *annotation retention policies*. A retention policy determines at what point an annotation is discarded. Java defines three such policies, which are encapsulated within the **java.lang.annotation.RetentionPolicy** enumeration. They are **SOURCE**, **CLASS**, and **RUNTIME**.

An annotation with a retention policy of **SOURCE** is retained only in the source file and is discarded during compilation.

An annotation with a retention policy of **CLASS** is stored in the **.class** file during compilation. However, it is not available through the JVM during run time.

An annotation with a retention policy of **RUNTIME** is stored in the **.class** file during compilation and is available through the JVM during run time. Thus, **RUNTIME** retention offers the greatest annotation persistence.

A retention policy is specified for an annotation by using one of Java's built-in annotations: **@Retention**. Its general form is shown here:

@Retention(*retention-policy*)

Here, *retention-policy* must be one of the previously discussed enumeration constants. If no retention policy is specified for an annotation, then the default policy of **CLASS** is used.

The following version of **MyAnno** uses **@Retention** to specify the **RUNTIME** retention policy. Thus, **MyAnno** will be available to the JVM during program execution.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}
```

```
public class Meta {

    public void doNothing(){};

    @MyAnnotation(str = "Annotation Example", val = 100)
    public static void myMethod(){
        try {
            Class<?> metaClass = Class.forName("annotation.Meta");
            Meta meta = (Meta) metaClass.getDeclaredConstructor().newInstance();
            Class<?> c = meta.getClass();
            Method method = c.getMethod("myMethod");
            MyAnnotation annotation = method.getAnnotation(MyAnnotation.class);
            System.out.printf("String %s, Val %d ", annotation.str(),
annotation.val());
        } catch (NoSuchMethodException |
                 ClassNotFoundException |
                 IllegalAccessException |
                 InstantiationException |
                 InvocationTargetException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Meta.myMethod();
    }
}
```

Using Default Values

You can give annotation members default values that will be used if no value is specified when the annotation is applied. A default value is specified by adding a **default** clause to a member's declaration. It has this general form:

type member( ) default *value*;

Here, *value* must be of a type compatible with *type*.

Here is **@MyAnno** rewritten to include default values:

```
// An annotation type declaration that includes defaults.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str() default "Testing";
  int val() default 9000;
}
```

Marker Annotation

```java
@Retention(RetentionPolicy.RUNTIME)
public @interface MyMarker {}

public class Meta {

    public void doNothing(){};

    @MyMarker
    public static void myMarker(){
        try {
            Class<?> metaClass = Class.forName("annotation.Meta");
            Meta meta = (Meta) metaClass.getDeclaredConstructor().newInstance();
            Method method = meta.getClass().getMethod("myMarker");
            System.out.println("Method is marked " +
method.isAnnotationPresent(MyMarker.class));

        } catch (ClassNotFoundException |
                InstantiationException |
                InvocationTargetException |
                NoSuchMethodException |
                IllegalAccessException classNotFoundException) {
            classNotFoundException.printStackTrace();
        }
    }
}
```

Single-Member Annotations

A *single-member* annotation contains only one member. It works like a normal annotation except that it allows a shorthand form of specifying the value of the member. When only one member is present, you can simply specify the value for that member when the annotation is applied—you don't need to specify the name of the member. However, in order to use this shorthand, the name of the member must be **value**.

Here is an example that creates and uses a single-member annotation:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// A single-member annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
  int value(); // this variable name must be value
}

class Single {

  // Annotate a method using a single-member annotation.
  @MySingle(100)
  public static void myMeth() {
    Single ob = new Single();
```

7. I/O Try-with-Resources

Streams

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to different types of devices. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the **java.io** package.

Byte Streams and Character Streams

Java defines two types of I/O streams: byte and character. *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated. Although old code that doesn't use character streams is becoming increasingly rare, it may still be encountered from time to time. As a general rule, old code should be updated to take advantage of character streams where appropriate.

One other point: at the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

An overview of both byte-oriented streams and character-oriented streams is presented in the following sections.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers. The non-deprecated byte stream classes in **java.io** are shown in Table 13-1. A few of these classes are discussed later in this section. Others are described in Part II of this book. Remember, to use the stream classes, you must import **java.io**.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that allows bytes to be returned to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The character stream classes in **java.io** are shown in Table 13-2.

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Transient key word

When we mark any variable as *transient,* then that variable is not serialized

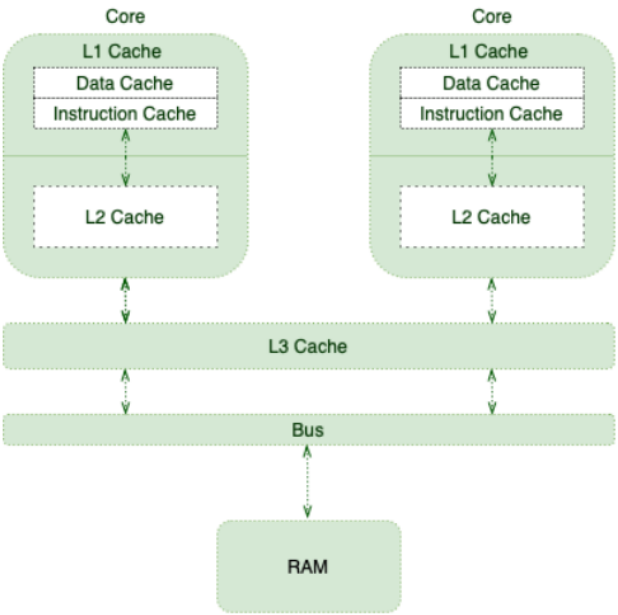
```
public class Book implements Serializable {
    private static final long serialVersionUID = -2936687026040726549L;
    private String bookName;
    private transient String description;
    private transient int copies;

    // getters and setters
}


Book book = new Book();
book.setBookName("Java Reference");
book.setDescription("will not be saved");
book.setCopies(25);


public static void serialize(Book book) throws Exception {
    FileOutputStream file = new FileOutputStream(fileName);
    ObjectOutputStream out = new ObjectOutputStream(file);
    out.writeObject(book);
    out.close();
    file.close();
}


public static Book deserialize() throws Exception {
    FileInputStream file = new FileInputStream(fileName);
    ObjectInputStream in = new ObjectInputStream(file);
    Book book = (Book) in.readObject();
    in.close();
    file.close();
    return book;
}


assertEquals("Java Reference", book.getBookName());
assertNull(book.getDescription()); // null
assertEquals(0, book.getCopies()); // null
```

Volatile key words

Processors are responsible for executing program instructions. Therefore, they need to retrieve both program instructions and required data from RAM.

As CPUs are capable of carrying out a significant number of instructions per second, fetching from RAM is not that ideal for them. To improve this situation, processors are using tricks like Out of Order Execution, Branch Prediction, Speculative Execution, and, of course, Caching.

This is where the following memory hierarchy comes into play:



In this simple example, we have two application threads: the main thread and the reader thread. Let's imagine a scenario in which the OS schedules those threads on two different CPU cores, where:

- The main thread has its copy of *ready* and *number* variables in its core cache
- The reader thread ends up with its copies, too
- The main thread updates the cached values

On most modern processors, write requests won't be applied right away after they're issued. In fact, **processors tend to queue those writes in a special write buffer**. After a while, they will apply those writes to main memory all at once.

With all that being said, **when the main thread updates the *number* and *ready* variables, there is no guarantee about what the reader thread may see. In other words, the reader thread may see the updated value right away, or with some delay, or never at all!**

This memory visibility may cause liveness issues in programs that are relying on visibility.

To ensure that updates to variables propagate predictably to other threads, we should apply the *volatile* modifier to those variables:

```
public class TaskRunner {

    private volatile static int number;
    private volatile static boolean ready;

    // same as before
}
```

This way, we communicate with runtime and processor to not reorder any instruction involving the *volatile* variable. Also, processors understand that they should flush any updates to these variables right away.

For multithreaded applications, we need to ensure a couple of rules for consistent behavior:

- Mutual Exclusion – only one thread executes a critical section at a time
- Visibility – changes made by one thread to the shared data are visible to other threads to maintain data consistency

synchronized methods and blocks provide both of the above properties, at the cost of application performance.

volatile is quite a useful keyword because it **can help ensure the visibility aspect of the data change without, of course, providing mutual exclusion**. Thus, it's useful in the places where we're ok with multiple threads executing a block of code in parallel, but we need to ensure the visibility property.

Static Import

```
// Use static import to bring sqrt() and pow() into view.
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
// Compute the hypotenuse of a right triangle.
class Hypot {
    public static void main(String[] args) {
        double side1, side2;
        double hypot;
        side1 = 3.0;
        side2 = 4.0;
        // Here, sqrt() and pow() can be called by themselves,
        // without their class name.
        hypot = sqrt(pow(side1, 2) + pow(side2, 2));
        System.out.println("Given sides of lengths " +
                side1 + " and " + side2 +
                " the hypotenuse is " + hypot);
    }
}
```

Invoking overloading constructor with this

- Without this

```
class MyClass {
    int a;
    int b;
    // initialize a and b individually
    MyClass(int i, int j) {
        a = i;
        b = j;
    }
    // initialize a and b to the same value
    MyClass(int i) {
        a = i;
        b = i;
    }
    // give a and b default values of 0
    MyClass( ) {
        a = 0;
        b = 0;
    }
}
```

- With this

```
class MyClass {
    int a;
    int b;
    // initialize a and b individually
    MyClass(int i, int j) {
        a = i;
        b = j;
    }
    // initialize a and b to the same value
    MyClass(int i) {
        this(i, i); // invokes MyClass(i, i)
    }
    // give a and b default values of 0
    MyClass( ) {
        this(0); // invokes MyClass(0)
    }
}
```

8. Modules

In its most fundamental sense, a *module* is a grouping of packages and resources that can be collectively referred to by the module's name. A *module declaration* specifies the name of a module and defines the relationship a module and its packages have to other modules.

Package	Module
A package cannot be deployed by itself	A module can be deployed by itself
A package groups together related classes	A module groups together related packages
Packages are present in Java right from the beginning	Modules were added by Java 9
Packages were added to keep related classes together and to allow developers to have a class with the same name in a different packages	Modules were added for security reasons and to reduce the size of the JDK
Classes defined within a package are accessible via reflection even if they are private	Classes defined within a module are not accessible outside the module via reflection
Packages do not require a package descriptor	Modules require a module descriptor which is a file called module-info.java

At the foundation of a module's capabilities are two key features. The first is a module's ability to specify that it requires another module. In other words, one module can specify that it *depends* on another. A dependence relationship is specified by use of a **requires** statement. By default, the presence of the required module is checked at both compile time and at run time. The second key feature is a module's ability to control which, if any, of its packages are accessible by another module. This is accomplished by use of the **exports** keyword. The public and protected types within a package are accessible to other modules only if they are explicitly exported. Here we will develop an example that introduces both of these features.

The following example creates a modular application that demonstrates some simple mathematical functions. Although this application is purposely very small, it illustrates the core concepts and procedures required to create, compile, and run module-based code. Furthermore, the general approach shown here also applies to larger, real-world applications. It is strongly recommended that you work through the example on your computer, carefully following each step.

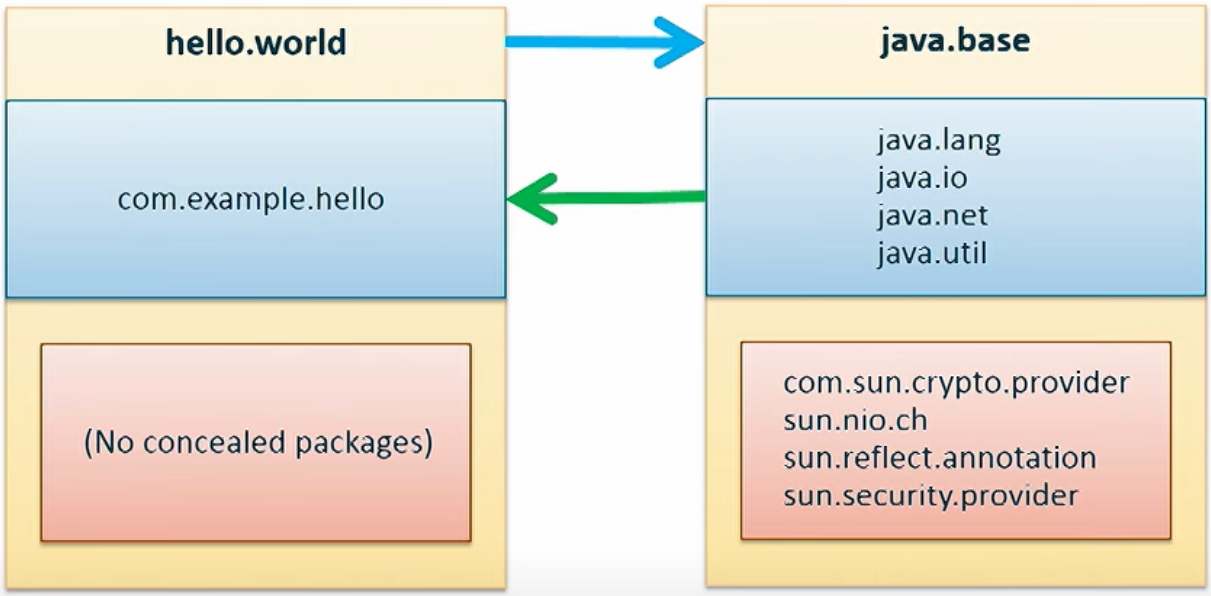
```
C:\Users\duyng>java --list-modules
java.base@11.0.12
java.compiler@11.0.12
java.datatransfer@11.0.12
java.desktop@11.0.12
java.instrument@11.0.12
java.logging@11.0.12
java.management@11.0.12
java.management.rmi@11.0.12
java.naming@11.0.12
java.net.http@11.0.12
java.prefs@11.0.12
java.rmi@11.0.12
java.scripting@11.0.12
```

Accessibility (JDK 1 – JDK 8)

- public
- protected
- package
- private

Accessibility (JDK 9)

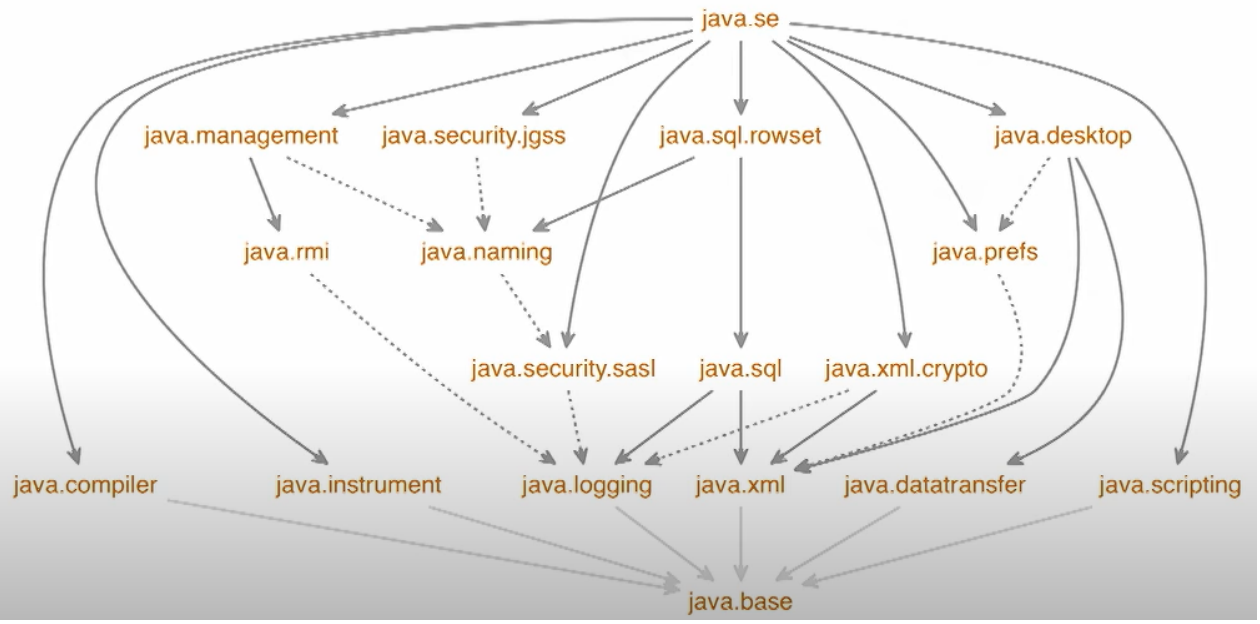
- *public to everyone*
- *public but only to friend modules*
- *public only within a module*
- protected
- package
- private

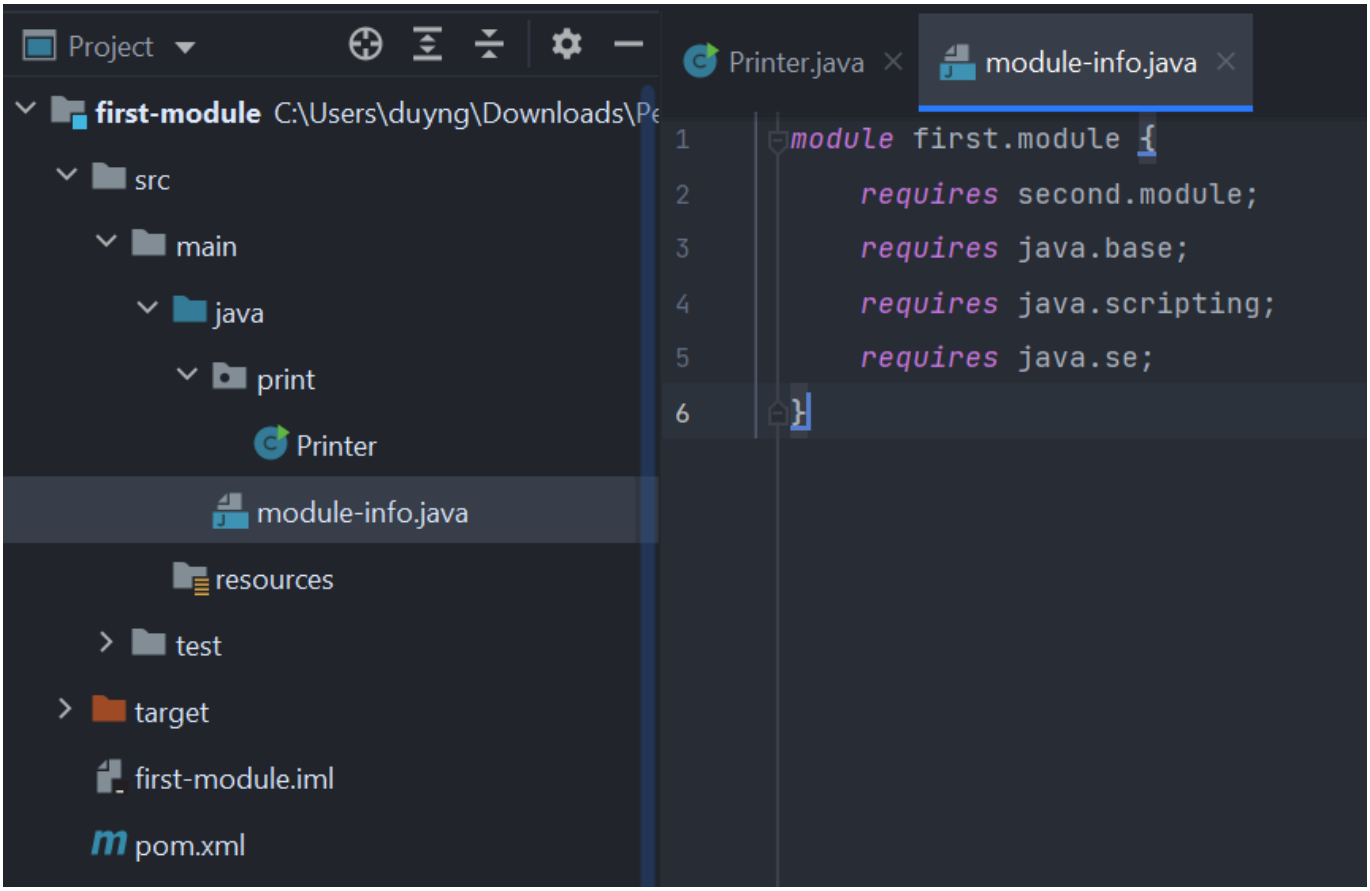


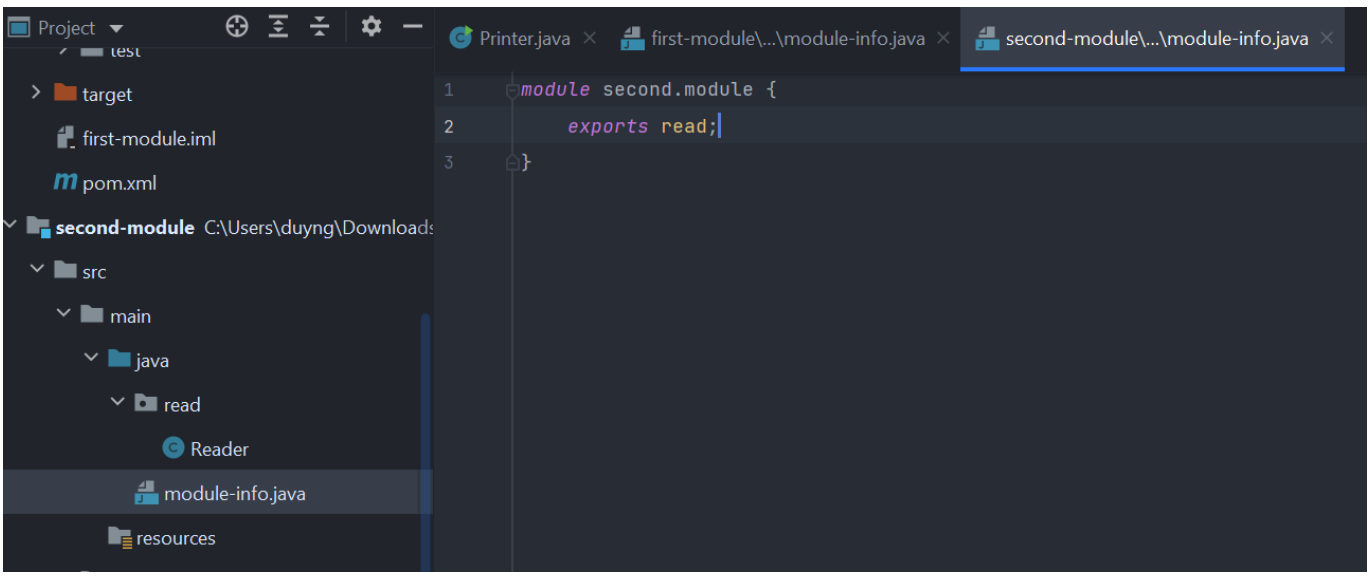

```
module hello.world {
    exports com.example.hello;
    requires java.base;
}
```

```
module java.base {
    exports java.lang;
    ...
}
```

The Modular JDK




Project
first-module C:\Users\duyng\Downloads\Pe
src
main
java
print
Printer
module-info.java
resources
test
target
first-module.iml
pom.xml
Printer.java
module-info.java
1 module first.module {
2 requires second.module;
3 requires java.base;
4 requires java.scripting;
5 requires java.se;
6 }


Project
target
first-module.iml
pom.xml
second-module C:\Users\duyng\Downloads
src
main
java
read
Reader
module-info.java
resources
Printer.java
first-module\...\module-info.java
second-module\...\module-info.java
1 module second.module {
2 exports read;
3 }

When a module exports a package, it makes all of the public and protected types in the package accessible to other modules. Furthermore, the public and protected members of those types are also accessible. However, if a package within a module is not exported, then it is private to that module, including all of its public types. For example, even though a class is declared as **public** within a package, if that package is not explicitly exported by an **exports** statement, then that class is not accessible to other modules. It is important to understand that the public and protected types of a package, whether exported or not, are always accessible within that package's module. The **exports** statement simply makes them accessible to outside modules. Thus, any nonexported package is only for the internal use of its module.

The key to understanding **requires** and **exports** is that they work together. If one module depends on another, then it must specify that dependence with **requires**. The module on which another depends must explicitly export (i.e., make accessible) the packages that the dependent module needs. If either side of this dependence relationship is missing, the dependent module will not compile. As it relates to the foregoing example, **MyModAppDemo** uses the functions in **SimpleMathFuncs**. As a result, the **appstart** module declaration contains a **requires** statement that names the **appfuncs** module. The **appfuncs** module declaration exports the **appfuncs.simplefuncs** package, thus making the public types in the **SimpleMathFuncs** class available. Since both sides of the dependence relationship have been fulfilled, the application can compile and run. If either is missing, the compilation will fail.

java.base and Platform modules

As mentioned at the start of this chapter, beginning with JDK 9 the Java API packages have been incorporated into modules. In fact, the modularization of the API is one of the primary benefits realized by the addition of the modules. Because of their special role, the API modules are referred to as *platform modules,* and their names all begin with the prefix **java**. Here are some examples: **java.base**, **java.desktop**, and **java.xml**. By modularizing the API, it becomes possible to deploy an application with only the packages that it requires, rather than the entire Java Runtime Environment (JRE). Because of the size of the full JRE, this is a very important improvement.

The fact that all of the Java API library packages are now in modules gives rise to the following question: How can the **main( )** method in **MyModAppDemo** in the preceding example use **System.out.println( )** without specifying a **requires** statement for the module that contains the **System** class? Obviously, the program will not compile and run unless **System** is present. The same question also applies to the use of the **Math** class in **SimpleMathFuncs**. The answer to this question is found in **java.base**.

Of the platform modules, the most important is **java.base**. It includes and exports those packages fundamental to Java, such as **java.lang**, **java.io**, and **java.util**, among many others. Because of its importance, **java.base** is *automatically accessible* to all modules. Furthermore, all other modules automatically require **java.base**. There is no need to include a **requires java.base** statement in a module declaration. (As a point of interest, it is not wrong to explicitly specify **java.base**, it's just not necessary.) Thus, in much the same way that **java.lang** is automatically available to all programs without the use of an **import** statement, the **java.base** module is automatically accessible to all module-based programs without explicitly requesting it.

Support for legacy code is provided by two key features. The first is the *unnamed module*. When you use code that is not part of a named module, it automatically becomes part of the unnamed module. The unnamed module has two important attributes. First, all of the packages in the unnamed module are automatically exported. Second, the unnamed module can access any and all other modules. Thus, when a program does not use modules, all API modules in the Java platform are automatically accessible through the unnamed module.

The second key feature that supports legacy code is the automatic use of the class path, rather than the module path. When you compile a program that does not use modules, the class path mechanism is employed, just as it has been since Java's original release. As a result, the program is compiled and run in the same way it was prior to the advent of modules.

9. Strings handling

Somewhat unexpectedly, when you create a **String** object, you are creating a string that cannot be changed. That is, once a **String** object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction. However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.

The **String**, **StringBuffer**, and **StringBuilder** classes are defined in **java.lang**. Thus, they are available to all programs automatically. All are declared **final**, which means that none of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations. All three implement the **CharSequence** interface.

Index	*String*	*String Buffer*	*String Builder*
Storage Area	Constant String Pool	Heap	Heap
Modifiable	No (immutable	Yes(mutable)	Yes(mutable)
Thread Safe	Yes	Yes	No
Thread Safe	Fast	Very slow	Fast

```java
public class StringHandler {

    public static void main(String[] args) {
        String string = "ABCD";
        char a = string.charAt(0);
        char[] chars = new char[string.length()];
        string.getChars(0, string.length(), chars, 0);
        System.out.println("Get chars" + Arrays.toString(chars)); // Get chars[A, B,
C, D]

        byte[] bytes = string.getBytes(StandardCharsets.UTF_8);
        System.out.println("Get bytes " + Arrays.toString(bytes)); // Get bytes [65,
66, 67, 68]

        System.out.println("To char array " + Arrays.toString(string.toCharArray()));
// To char array [A, B, C, D]

        String str1 = "Hello, How are you";
        String str2 = "How";

        System.out.print("Result of Test1: " );
        System.out.println(str1.regionMatches(7, str2, 0, 3));

    }
}
```

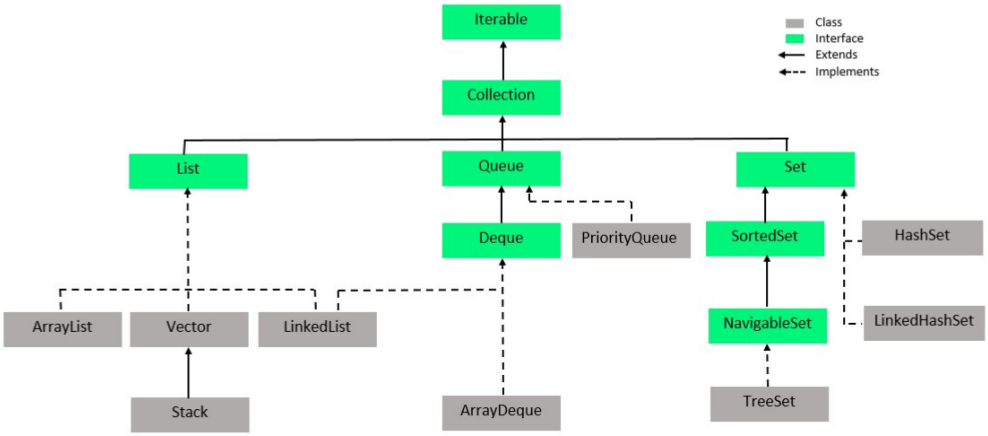
10. Collection frameworks

The Java Collections Framework standardizes the way in which groups of objects are handled by your programs. Collections were not part of the original Java release, but were added by J2SE 1.2. Prior to the Collections Framework, Java provided ad hoc classes such as **Dictionary**, **Vector**, **Stack**, and **Properties** to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. The way that you used **Vector** was different from the way that you used **Properties**, for example. Also, this early, ad hoc approach was not designed to be easily extended or adapted. Collections were an answer to these (and other) problems.

Algorithms are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the **Collections** class. Thus, they are available for all collections. Each collection class need not implement its own versions. The algorithms provide a standard means of manipulating collections.

Another item closely associated with the Collections Framework is the **Iterator** interface. An *iterator* offers a general-purpose, standardized way of accessing the elements within a collection, one at a time. Thus, an iterator provides a means of *enumerating the contents of a collection*. Because each collection provides an iterator, the elements of any collection class can be accessed through the methods defined by **Iterator**. Thus, with only small changes, the code that cycles through a set can also be used to cycle through a list, for example.

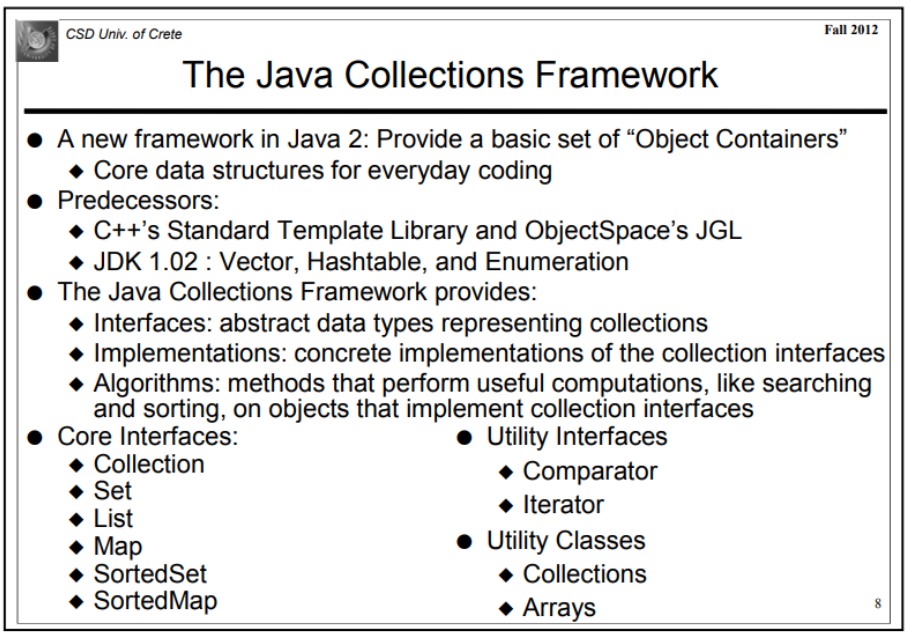
JDK 8 added another type of iterator called a *spliterator*. In brief, spliterators are iterators that provide support for parallel iteration. The interfaces that support spliterators are **Spliterator** and several nested interfaces that support primitive types. Also available are iterator interfaces designed for use with primitive types, such as **PrimitiveIterator** and **PrimitiveIterator.OfDouble**.



List

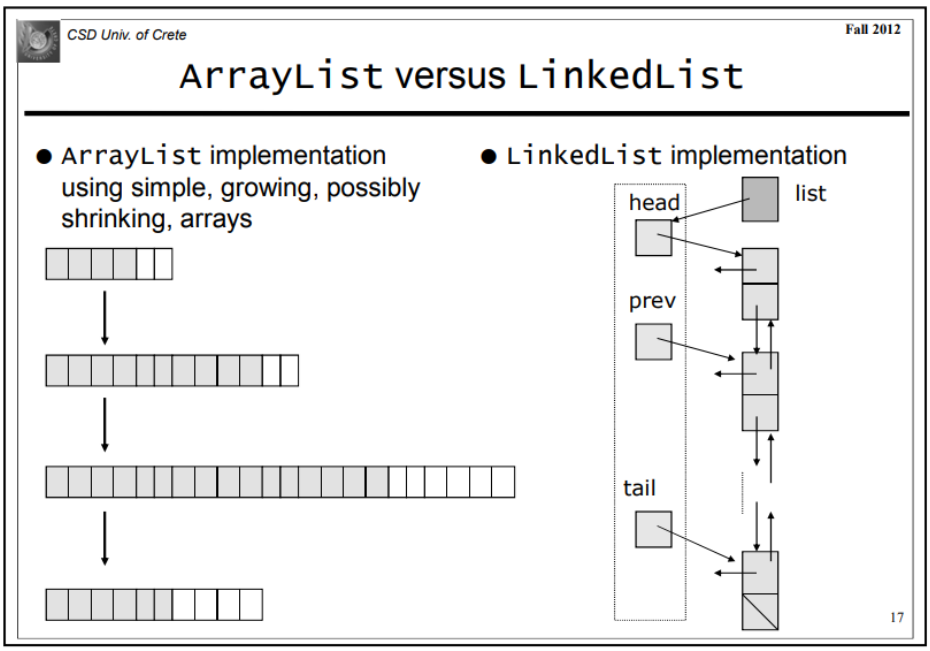
For operation require add and remove in the middle of the List **LinkedList** is the better choide. While for operation require read from index **ArrayList** is the better candidate

Algorithm	array ArrayList	LinkedList
access front	O(1)	O(1)
access back	O(1)	O(1)
access middle	O(1)	O(N)
insert at front	O(N)	O(1)
insert at back	O(1)	O(1)
insert in middle	O(N)	O(1)

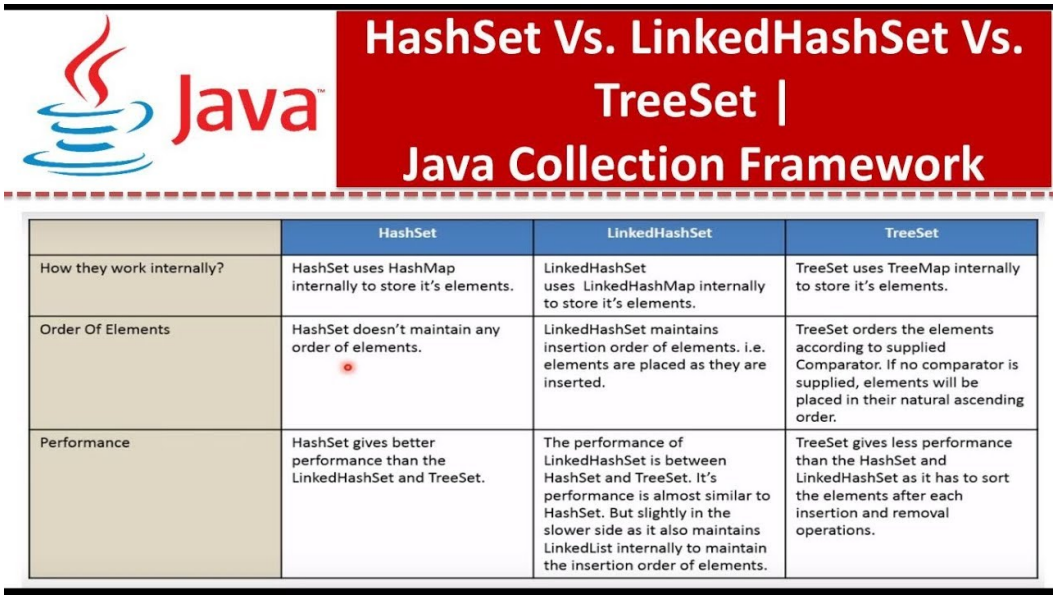


LIST VERSUS SET

LIST	SET
A sub-interface of Collection that contains methods to perform operations such as insert and delete based on the index	A sub-interface of Collection that contains methods to perform operations such as insert and delete elements while maintaining unique elements
Stores duplicate values	Does not store duplicate values
ArrayList, LinkedList, and Vector implement the List interface	HashSet, LinkedHashSet and TreeSet classes implement the Set interface
It is possible to use an Iterator or a ListIterator to traverse the items in the List	It is not possible to use the ListIterator to traverse the items in the Set
Can have many null values	Can only have a single null value



Set



	HashSet	LinkedHashSet	TreeSet
How they work internally?	HashSet uses HashMap internally to store it's elements.	LinkedHashSet uses LinkedHashMap internally to store it's elements.	TreeSet uses TreeMap internally to store it's elements.
Order Of Elements	HashSet doesn't maintain any order of elements.	LinkedHashSet maintains insertion order of elements. i.e. elements are placed as they are inserted.	TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order.
Performance	HashSet gives better performance than the LinkedHashSet and TreeSet.	The performance of LinkedHashSet is between HashSet and TreeSet. It's performance is almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList internally to maintain the insertion order of elements.	TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the elements after each insertion and removal operations.

Implementation of Set

```
public class SetTest {
    public static void main(String[] args) {
        LocalDateTime start = LocalDateTime.now();
        System.out.println("Testing Hash Set");
        HashSet<String> hashSet = new HashSet<>();
        hashSet.add("ACB");
        hashSet.add("OCB");
        hashSet.add("ECB");
        hashSet.add("KLB");
        System.out.print("[ - ");
        for (String s : hashSet) {
            System.out.print(s);
            System.out.print(" - ");
        }
        System.out.print("]");
        LocalDateTime end = LocalDateTime.now();
        Duration duration = Duration.between(end, start);
        System.out.println();
        System.out.println("Performance " + duration.toNanos() + " ns");
        /*
            Testing Hash Set
            [ - ACB - KLB - ECB - OCB - ]
            Performance -1000300 ns

        -> Order don't preserve in case of Hash Set
        */
    }
}
```

LinkedHashSet implementation

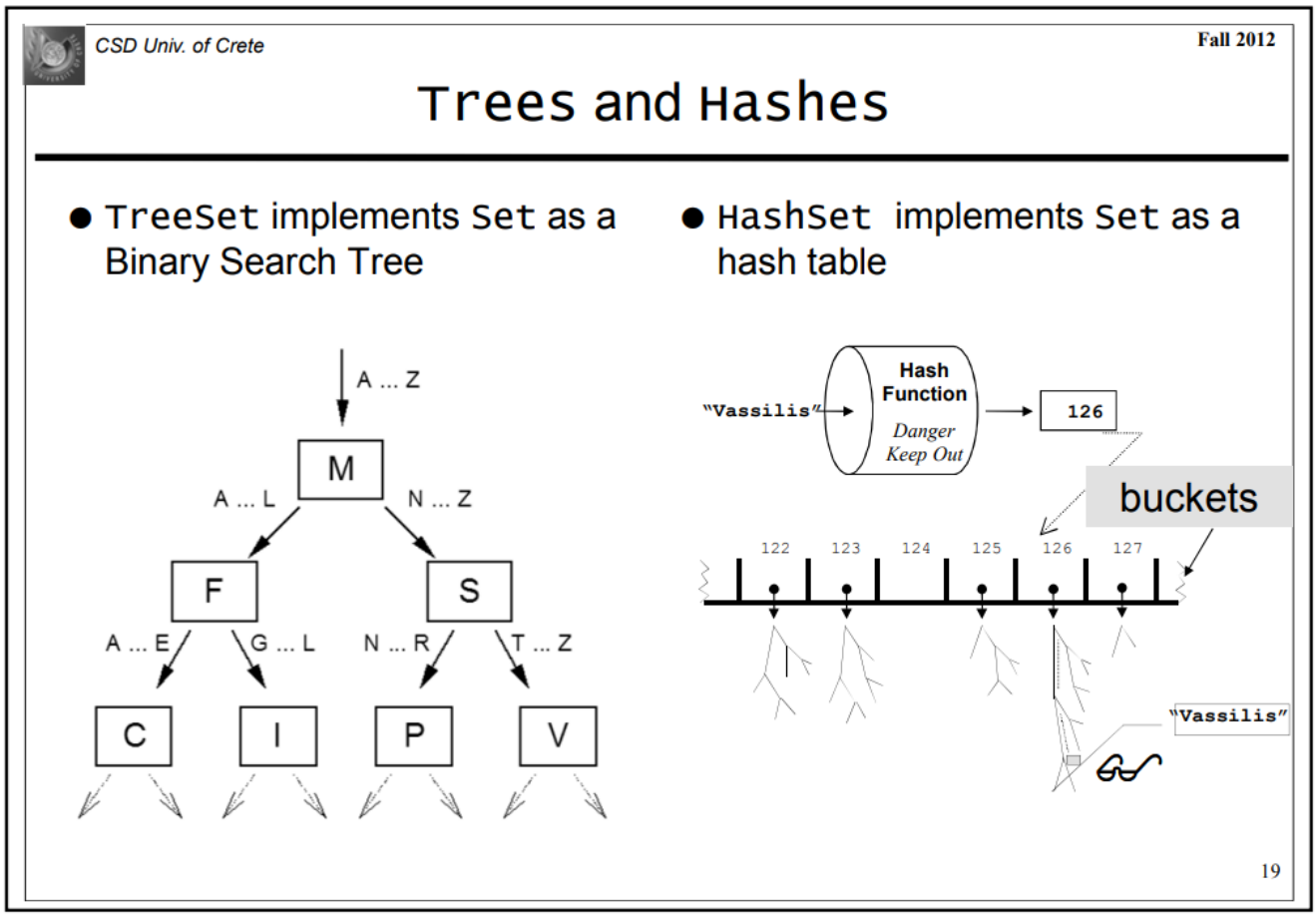
```
public class LinkedHashSetTest {
    public static void main(String[] args) {
        LinkedHashSet<String> hashSet = new LinkedHashSet<>();
        hashSet.add("ACB");
        hashSet.add("OCB");
        hashSet.add("ECB");
        hashSet.add("KLB");

        LocalDateTime start = LocalDateTime.now();
        System.out.print("[ - ");
        for (String s : hashSet) {
            System.out.print(s);
            System.out.print(" - ");
        }
        System.out.print("]");
        LocalDateTime end = LocalDateTime.now();
        Duration duration = Duration.between(end, start);
        System.out.println();
        System.out.println("Performance " + duration.toNanos() + " ns");
        /*
            [ - ACB - OCB - ECB - KLB - ]
            Performance -998600 ns

            -> LinkedHashSet Retain order
         */
    }
}
```

TreeSet implementation

```
public class TreeSetTest {
    public static void main(String[] args) {
        Comparator<String> comparator = Comparator
                                        .comparing(s -> s.charAt(0));
        TreeSet<String> treeSet = new TreeSet<>(comparator);
        treeSet.add("2A");
        treeSet.add("1C");
        treeSet.add("3D");
        for (String s : treeSet){
            System.out.println(s);
        }
        /* 1C -> 2A -> 3D */
    }
}
```

Conclusion:

- **HashSet**: don't keep order
- **LinkedHashSet**: keep natural order
- **TreeSet**: customize order



Queue, Dequeue, PriorityQueue

DEQUE	STACK	QUEUE
size()	size()	size()
isEmpty()	isEmpty()	isEmpty()
Insert_First()	-	-
Insert_Last()	Push()	Enqueue()
Remove_First()	-	Dequeue()
Remove_Last()	Pop()	-

ArrayQueue implementation:

```
public class ArrayQueueTest {
    public static void main(String[] args) {
        ArrayDeque<String> arrayDeque = new ArrayDeque<>();
        arrayDeque.add("111");
        arrayDeque.add("222");
        arrayDeque.add("333");

        System.out.println("First object be remove is " + arrayDeque.remove());
        // 111 because of FIFO
    }
}
```

PriorityQueue implementation

```
public class PriorityQueueTest {

    public static void main(String[] args) {
        Comparator<String> comparator = Comparator.comparing(String::length);
        PriorityQueue<String> queue = new PriorityQueue<>(comparator);

        queue.add("4444");
        queue.add("22");
        queue.add("333");

        System.out.println("First object be remove is " + queue.remove());
        // 22 because of lowest length
    }
}
```

Map

- LinkedHashMap
- TreeMap
- HashMap

Property	HashMap	LinkedHashMap	TreeMap
Time Complexity (Big O notation) Get, Put, ContainsKey and Remove method	O(1)	O(1)	O(log n)
Iteration Order	Random	Sorted according to either Insertion Order or Access Order (as specified during construction)	Sorted according to either natural order of keys or comparator (as specified during construction)
Null Keys	allowed	allowed	Not allowed if Key uses Natural Ordering or Comparator does not support comparison on null Keys
Interface	Map	Map	Map, SortedMap and NavigableMap
Synchronization	none, use Collections.synchronizedMap()	None, use Collections.synchronizedMap()	none, use Collections.synchronizedMap()
Data Structure	List of Buckets, if more than 8 entries in bucket then Java 8 will switch to balanced tree from linked list	Doubly Linked List of Buckets	Red-Black Tree (a kind of self-balancing binary search tree) implementation of Binary Tree. This data structure offers O (log n) for Insert, Delete and Search operations and O (n) Space Complexity
Applications	General Purpose, fast retrieval, non-synchronized. ConcurrentHashMap can be used where concurrency is involved.	Can be used for LRU cache, other places where insertion or access order matters	Algorithms where Sorted or Navigable features are required. For example, find among the list of employees whose salary is next to given employee, Range Search, etc.
Requirements for Keys	Equals() and hashCode() needs to be overwritten	Equals() and hashCode() needs to be overwritten	Comparator needs to be supplied for Key implementation, otherwise natural order will be used to sort the keys

HashMap (Thread-safe verision = **ConcurrentHashMap**) implementation:

```
public class HashMapTest {

    public static void main(String[] args) {
        HashMap<String, String> map = new HashMap<>();
        map.put("key-A", "value-A");
        map.put("key-B", "value-B");
        map.put("key-C", "value-C");

        for(Map.Entry<String, String> entry : map.entrySet()){
            System.out.println(entry);
        }
        /*
            key-A=value-A
            key-C=value-C
            key-B=value-B
         */
    }
}
```

LinkedHashMap test:

```
public class LinkedHashMapTest {
    public static void main(String[] args) {
        LinkedHashMap<String, String> map = new LinkedHashMap<>();
        map.put("key-A", "value-A");
        map.put("key-B", "value-B");
        map.put("key-C", "value-C");

        for(Map.Entry<String, String> entry : map.entrySet()){
            System.out.println(entry);
        }
        /*
            key-A=value-A
            key-B=value-B
            key-C=value-C

            -> Order is remain
         */
    }
}
```

TreeHashMap

```java
public class TreeMapTest {
    public static void main(String[] args) {
        Comparator<String> comparator = Comparator.comparing(String::length);
        TreeMap<String, String> map = new TreeMap<>(comparator);
        map.put("333", "value-C");
        map.put("22", "value-B");
        map.put("1", "value-A");

        for(Map.Entry<String, String> entry : map.entrySet()){
            System.out.println(entry);
        }

        /*
            1=value-A
            22=value-B
            333=value-C
         */
    }
}
```

11. Exploring NIO

NIO Fundamentals

The NIO system is built on two foundational items: buffers and channels. A *buffer* holds data. A *channel* represents an open connection to an I/O device, such as a file or a socket. In general, to use the NIO system, you obtain a channel to an I/O device and a buffer to hold data. You then operate on the buffer, inputting or outputting data as needed. The following sections examine buffers and channels in more detail.

Buffers

Buffers are defined in the **java.nio** package. All buffers are subclasses of the **Buffer** class, which defines the core functionality common to all buffers: current position, limit, and capacity. The *current position* is the index within the buffer at which the next read or write operation will take place. The current position is advanced by most read or write operations. The *limit* is the index value one past the last valid location in the buffer. The *capacity* is the number of elements that the buffer can hold. Often the limit equals the capacity of the buffer. **Buffer** also supports mark and reset. **Buffer** defines several methods, which are shown in Table 23-1.

Channels

Channels are defined in **java.nio.channels**. A channel represents an open connection to an I/O source or destination. Channels implement the **Channel** interface. It extends **Closeable**, and it extends **AutoCloseable**. By implementing **AutoCloseable**, channels can be managed with a **try**-with-resources statement. When used in a **try**-with-resources block, a channel is closed automatically when it is no longer needed. (See Chapter 13 for a discussion of **try**-with-resources.)

One way to obtain a channel is by calling **getChannel( )** on an object that supports channels. For example, **getChannel( )** is supported by the following I/O classes:

DatagramSocket	FileInputStream	FileOutputStream
RandomAccessFile	ServerSocket	Socket

The specific type of channel returned depends upon the type of object **getChannel( )** is called on. For example, when called on a **FileInputStream**, **FileOutputStream**, or **RandomAccessFile**, **getChannel( )** returns a channel of type **FileChannel**. When called on a **Socket**, **getChannel( )** returns a **SocketChannel**.

Another way to obtain a channel is to use one of the **static** methods defined by the **Files** class. For example, using **Files**, you can obtain a byte channel by calling **newByteChannel( )**. It returns a **SeekableByteChannel**, which is an interface implemented by **FileChannel**. (The **Files** class is examined in detail later in this chapter.)

Charsets and Selectors

Two other entities used by NIO are charsets and selectors. A *charset* defines the way that bytes are mapped to characters. You can encode a sequence of characters into bytes using an *encoder*. You can decode a sequence of bytes into characters using a *decoder*. Charsets, encoders, and decoders are supported by classes defined in the **java.nio.charset** package. Because default encoders and decoders are provided, you will not often need to work explicitly with charsets.

A *selector* supports key-based, non-blocking, multiplexed I/O. In other words, selectors enable you to perform I/O through multiple channels. Selectors are supported by classes defined in the **java.nio.channels** package. Selectors are most applicable to socket-backed channels.

We will not use charsets or selectors in this chapter, but you might find them useful in your own applications.

Code Sample

```
public class NioTest {
    static void readFileFromChannel(){
        int count;
        Path filePath = null;

        try {
            filePath = Path.of("text.txt");
        } catch (InvalidPathException e){
            e.printStackTrace();
            return;
        }

        //
        try (SeekableByteChannel channel = Files.newByteChannel(filePath) ){
            ByteBuffer buffer = ByteBuffer.allocate(128);
            do {
                count = channel.read(buffer);

                if (count != -1){
                    buffer.rewind();
                    for (int i = 0; i < count; i++) {
                        System.out.print((char) buffer.get());
                    }
                }
            } while (count != -1);
            System.out.println();
        } catch (IOException exception) {
            exception.printStackTrace();
        }

        // Using Map Buffer
        try (FileChannel channel = (FileChannel) Files
                                    .newByteChannel(filePath)){
            long fSize = channel.size();
            MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY,
                                                  0, fSize);
            for (int i = 0; i < fSize; i++) {
                System.out.print((char) buffer.get());
            }
            System.out.println();
        } catch (IOException exception) {
            exception.printStackTrace();
        }
    }

    static void writeToFileFromChannel(){
        Path filePath = Path.of("text.txt");
        StandardOpenOption[] options = new StandardOpenOption[]{
                                        StandardOpenOption.WRITE,
                                        StandardOpenOption.READ};
```

```
        try (FileChannel channel = (FileChannel) Files.newByteChannel(filePath,
options)){
            MappedByteBuffer mBuf = channel.map(FileChannel.MapMode.READ_WRITE,
                                        0, 26);
            // Write some bytes to the buffer.
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

        } catch (IOException exception) {
            exception.printStackTrace();
        }
    }

    static void copyFileNIO(){
        try {
            Path source = Path.of("text.txt");
            Path target = Path.of("dest.txt");
            // Copy the file.
            Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch (IOException e) {
            System.out.println("I/O Error " + e);
        }
    }

    static void pathFileAndSystemOperation(){
        Path filepath = Path.of("text.txt");
        System.out.println("File Name: " + filepath.getName(0));
        System.out.println("Path: " + filepath);
        System.out.println("Absolute Path: " + filepath.toAbsolutePath());
        System.out.println("Parent: " + filepath.getParent());
        if(Files.exists(filepath))
            System.out.println("File exists");
        else
            System.out.println("File does not exist");
        try {
            if(Files.isHidden(filepath))
                System.out.println("File is hidden");
            else
                System.out.println("File is not hidden");
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
        }
        Files.isWritable(filepath);
        System.out.println("File is writable");
        Files.isReadable(filepath);
        System.out.println("File is readable");
        try {
            BasicFileAttributes attribs =
                    Files.readAttributes(filepath, BasicFileAttributes.class);
```

```
            if(attribs.isDirectory())
                System.out.println("The file is a directory");
            else
                System.out.println("The file is not a directory");
            if(attribs.isRegularFile())
                System.out.println("The file is a normal file");
            else
                System.out.println("The file is not a normal file");
            if(attribs.isSymbolicLink())
                System.out.println("The file is a symbolic link");
            else
                System.out.println("The file is not a symbolic link");
            System.out.println("File last modified: " + attribs.lastModifiedTime());
            System.out.println("File size: " + attribs.size() + " Bytes");
        } catch(IOException e) {
            System.out.println("Error reading attributes: " + e);
        }
    }

    public static void main(String[] args) {
        NioTest.writeToFileFromChannel();
        NioTest.readFileFromChannel();
        NioTest.copyFileNIO();
        NioTest.pathFileAndSystemOperation();
    }
}
```

12. Networking

java.net Networking classes and interfaces:

Authenticator	InetAddress	SocketAddress
CacheRequest	InetSocketAddress	SocketImpl
CacheResponse	InterfaceAddress	SocketPermission
ContentHandler	JarURLConnection	StandardSocketOption
CookieHandler	MulticastSocket	UnixDomainSocketAddress

CookieManager	NetPermission	URI
DatagramPacket	NetworkInterface	URL
DatagramSocket	PasswordAuthentication	URLClassLoader
DatagramSocketImpl	Proxy	URLConnection
HttpCookie	ProxySelector	URLDecoder
HttpURLConnection	ResponseCache	URLEncoder
IDN	SecureCacheResponse	URLPermission
Inet4Address	ServerSocket	URLStreamHandler
Inet6Address	Socket	

```
static void inetTest() throws UnknownHostException {
        InetAddress address = InetAddress.getLocalHost();
        System.out.println(address);

        address = InetAddress.getByName("www.HerbSchildt.com");
        System.out.println(address);

        InetAddress[] SW = InetAddress.getAllByName("www.nba.com");
        for (InetAddress inetAddress : SW)
            System.out.println(inetAddress);

        /*
            DESKTOP-HLTRTU1/192.168.0.182
            www.HerbSchildt.com/216.92.65.4
            www.nba.com/23.66.4.57
         */
    }
```

Inet4Address and Inet6Address

Java includes support for both IPv4 and IPv6 addresses. Because of this, two subclasses of **InetAddress** were created: **Inet4Address** and **Inet6Address**. **Inet4Address** represents a traditional-style IPv4 address. **Inet6Address** encapsulates a newer IPv6 address. Because they are subclasses of **InetAddress,** an **InetAddress** reference can refer to either. This is one way that Java was able to add IPv6 functionality without breaking existing code or adding many more classes. For the most part, you can simply use **InetAddress** when working with IP addresses because it can accommodate both styles.

TCP/IP Client Sockets

TCP/IP Client Sockets

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet, subject to security constraints.

There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The **ServerSocket** class is designed to be a "listener," which waits for clients to connect before doing anything. Thus, **ServerSocket** is for servers. The **Socket** class is for clients. It is designed to connect to server sockets and initiate protocol exchanges. Because client sockets are the most commonly used by Java applications, they are examined here.

```
static void testSocketClient(){
    int c;
    try {
        Socket s = new Socket("127.0.0.1", 9980);
        InputStream inputStream = s.getInputStream();
        OutputStream outputStream = s.getOutputStream();

        outputStream.write("Hello".getBytes());
        while((c = inputStream.read()) != -1){
            System.out.println((char) c);
        }
        s.close();
    } catch (IOException exception) {
        exception.printStackTrace();
    }
}
```

URL

```
URL hp = new URL("http://www.HerbSchildt.com/WhatsNew");
System.out.println("Protocol: " + hp.getProtocol());
System.out.println("Port: " + hp.getPort());
System.out.println("Host: " + hp.getHost());
System.out.println("File: " + hp.getFile());
System.out.println("Ext:" + hp.toExternalForm());
```

13. Some useful classes in java.lang

Runtime classes

```
    public static void main(String[] args) {
        Runtime r = Runtime.getRuntime();
        Process p = null;
        try {
            p = r.exec("notepad");
        } catch (Exception e) {
            System.out.println("Error executing notepad.");
        }
    }
```

Environment Properties

file.separator	java.vendor	java.vm.version
java.class.path	java.vendor.url	line.separator
java.class.version	java.vendor.version	native.encoding
java.compiler	java.version	os.arch
java.home	java.version.date	os.name
java.io.tmpdir	java.vm.name	os.version
java.library.path	java.vm.specification.name	path.separator
java.specification.name	java.vm.specification.vendor	user.dir
java.specification.vendor	java.vm.specification.version	user.home
java.specification.version	java.vm.vendor	user.name

You can obtain the values of various environment variables by calling the **System.getProperty( )** method. For example, the following program displays the path to the current user directory:

```
class ShowUserDir {
  public static void main(String[] args) {
    System.out.println(System.getProperty("user.dir"));
  }
}
```

Deep Copy

Let's add the *clone()* method to the *Address* class:

```
@Override
public Object clone() {
    try {
        return (Address) super.clone();
    } catch (CloneNotSupportedException e) {
        return new Address(this.street, this.getCity(), this.getCountry());
    }
}
```

Now let's implement *clone()* for the *User* class:

```
@Override
public Object clone() {
    User user = null;
    try {
        user = (User) super.clone();
    } catch (CloneNotSupportedException e) {
        user = new User(
          this.getFirstName(), this.getLastName(), this.getAddress());
    }
    user.address = (Address) this.address.clone();
    return user;
}
```

```
@Test
public void whenModifyingOriginalObject_thenCommonsCloneShouldNotChange() {
    Address address = new Address("Downing St 10", "London", "England");
    User pm = new User("Prime", "Minister", address);
    User deepCopy = (User) SerializationUtils.clone(pm);

    address.setCountry("Great Britain");

    assertThat(deepCopy.getAddress().getCountry())
      .isNotEqualTo(pm.getAddress().getCountry());
}
```

14. Reflection

In Java, it is possible to inspect fields, classes, methods, annotations, interfaces, etc. at runtime. You do not need to know how classes or methods are called, neither the parameters that are needed, all of that can be retrieved at runtime using reflection. It is also possible to instantiate new classes, to create new instances and to execute their methods, all of it using reflection.

Reflection is present in Java since the beginning of the times via its reflection API. The class `Class` contains all the reflection related methods that can be applied to classes and objects like the ones that allow a programmer to retrieve the class name, to retrieve the public methods of a class, etc. Other important classes are `Method`, `Field` and `Type` containing specific reflection methods that we are going to see in this tutorial.

Although reflection is very useful in many scenarios, it should not be used for everything. If some operation can be executed without using reflection, then we should not use it. Here are some reasons:

- The performance is affected by the use of reflection since all compilation optimizations cannot be applied: reflection is resolved at runtime and not at compile stages.
- Security vulnerabilities have to be taken into consideration since the use of reflection may not be possible when running in secure contexts like Applets.
- Another important disadvantage that is good to mention here is the maintenance of the code. If your code uses reflection heavily it is going to be more difficult to maintain. The classes and methods are not directly exposed in the code and may vary dynamically so it can get difficult to change the number of parameters that a method expects if the code that calls this method is invoked via reflection.
- Tools that automatically refactor or analyze the code may have trouble when a lot of reflection is present.

Despite all the limitations, reflection is a very powerful tool in Java that can be taken into consideration in several scenarios.

In general, reflection can be used to observe and modify the behavior of a program at runtime. Here is a list with the most common use cases:

- IDEs can heavily make use of reflection in order to provide solutions for auto completion features, dynamic typing, hierarchy structures, etc. For example, IDEs like Eclipse or PHP Storm provide a mechanism to retrieve dynamically the arguments expected for a given method or a list of public methods starting by "get" for a given instance. All these are done using reflection.
- Debuggers use reflection to inspect dynamically the code that is being executed.
- Test tools like Junit or Mockito use reflection in order to invoke desired methods containing specific syntax or to mock specific classes, interfaces and methods.
- Dependency injection frameworks use reflection to inject beans and properties at runtime and initialize all the context of an application.
- Code analysis tools like PMD or Findbugs use reflection in order to analyze the code against the list of code violations that are currently configured.
- External tools that make use of the code dynamically may use reflection as well.

Reflection components and mechanisms

```
static void testClassReflection() throws ClassNotFoundException,
                                         NoSuchMethodException,
                                         IllegalAccessException,
                                         InvocationTargetException,
                                         InstantiationException {
    String stringer = "Hello";
    Class<?> stringGetClass = Class.forName( "java.lang.String" );
    String newInstanceStringClass = (String) stringGetClass
                                             .getConstructor(String.class)
                                             .newInstance("hello");
    System.out.println("newInstanceStringClass " + newInstanceStringClass);
}
```

Dummy Class & Inspect fields

```
class Dummy {
    int value;
    void testMethod(){
        System.out.println("Do nothing");
    }
}
```

```
static void testFields() {
    Class<Dummy> dummyClass = Dummy.class;
    Field[] fields = dummyClass.getDeclaredFields();
    for( Field field : fields )
    {
        System.out.println( "************************" );
        System.out.println( "Name: " + field.getName() );
        System.out.println( "Type: " + field.getType() );
        System.out.println( "Modifiers:" + field.getModifiers() );
    }
}
```

Dynamic Proxies

```
class HandlerImpl implements InvocationHandler{

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
                    throws Throwable {

        System.out.println( "using proxy " + proxy.getClass().getName() );
        System.out.println( "method " + method.getName() + " from interface "
                            + method.getDeclaringClass().getName() );

        if (method.getDeclaringClass()
                  .getName()
                  .equals("reflection.Dummy")){
            Dummy dummy = Dummy.class.getDeclaredConstructor().newInstance();
            return method.invoke(dummy, args);
        }
        return null;
    }
}

public class DynamicProxyTest {
    public static void main(String[] args) throws Exception {
        InvocationHandler handler = new HandlerImpl();
        Dummy proxy = (Dummy) Proxy.newProxyInstance(Dummy.class.getClassLoader(),
                                    Dummy.class.getInterfaces(), handler);
        proxy.testMethod();
    }
}
```

15. Concurrency Utilities

15.1 Concurrency Design Principles

Users of computer systems are always looking for better performance of their systems. They want to get higher quality videos, better video games, and faster network speeds. Some years ago, processors gave better performance to users by increasing their speed. But now, processors don't increase their speed. Instead of this, they add more cores so that the operating system can execute more than one task at a time. This is called **concurrency**. Concurrent programming includes all the tools and techniques to have multiple tasks or processes running at the same time in a computer, communicating and synchronizing between them without data loss or inconsistency. In this chapter, we will cover the following topics:

- Basic concurrency concepts
- Possible problems in concurrent applications
- A methodology to design concurrent algorithms
- Java Concurrency API
- Concurrency design patterns
- Tips and tricks to design concurrency algorithms

Concurrency versus parallelism

Concurrency and parallelism are very similar concepts. Different authors give different definitions for these concepts. The most accepted definition talks about concurrency as being when you have more than one task in a single processor with a single core. In this case, the operating system's task scheduler quickly switches from one task to another, so it seems that all the tasks run simultaneously. The same definition talks about parallelism as being when you have more than one task running simultaneously on different computers, processors, or cores inside a processor.

Another definition talks about concurrency being when you have more than one task (different tasks) that run simultaneously on your system. Yet another definition discusses parallelism as being when you have different instances of the same task that run simultaneously over different parts of a dataset.

The last definition talks about parallelism being when you have more than one task that runs simultaneously in your system and talks about concurrency as a way to explain the different techniques and mechanisms the programmer has to synchronize with the tasks and their access to shared resources.

As you can see, both concepts are very similar, and this similarity has increased with the development of multicore processors.

Synchronization

In concurrency, we can define **synchronization** as the coordination of two or more tasks to get the desired results. We have two kinds of synchronization:

- **Control synchronization**: When, for example, one task depends on the end of another task, the second task can't start before the first has finished
- **Data access synchronization**: When two or more tasks have access to a shared variable and only one of the tasks can access the variable

A concept closely related to synchronization is **critical section**. A critical section is a piece of code that can be only executed by one task at a time because of its access to a shared resource. **Mutual exclusion** is the mechanism used to guarantee this requirement and can be implemented in different ways.

Keep in mind that synchronization helps you avoid some errors you might have with concurrent tasks (they will be described later in this chapter), but it introduces some overhead to your algorithm. You have to calculate the number of tasks very carefully, which can be performed independently without intercommunication you will have in your parallel algorithm. It's the **granularity** of your concurrent algorithm. If you have a **coarse-grained granularity** (big tasks with low intercommunication), the overhead due to synchronization will be low. However, maybe you won't benefit from all the cores of your system. If you have a **fine-grained granularity** (small tasks with high intercommunication), the overhead due to synchronization will be high, and perhaps the throughput of your algorithm won't be good.

There are different mechanisms to get synchronization in a concurrent system. The most popular mechanisms from a theoretical point of view are:

- **Semaphore**: A semaphore is a mechanism that can be used to control the access to one or more units of a resource. It has a variable that stores the number of resources that can be used and two atomic operations to manage the value of the variable. A **mutex** (short for **mutual exclusion**) is a special kind of semaphore that can take only two values (*resource is free* and *resource is busy*), and only the process that sets the mutex to *busy* can release it. A mutex can help you to avoid race conditions by protecting a critical section.
- **Monitor**: A monitor is a mechanism to get mutual exclusion over a shared resource. It has a mutex, a condition variable, and two operations to wait for the condition and signal the condition. Once you signal the condition, only one of the tasks that are waiting for it continues with its execution.

The last concept related to synchronization you're going to learn in this chapter is **thread safety**. A piece of code (or a method or an object) is **thread-safe** if all the users of shared data are protected by synchronization mechanisms. A non-blocking, **compare-and-swap (CAS)** primitive of the data is immutable, so you can use that code in a concurrent application without any problems.

Immutable object

An **immutable object** is an object with a very special characteristic. You can't modify its visible state (the value of its attributes) after its initialization. If you want to modify an immutable object, you have to create a new one.

Its main advantage is that it is thread-safe. You can use it in concurrent applications without any problem.

An example of an immutable object is the `String` class in Java. When you assign a new value to a `String` object, you are creating a new one.

Atomic operations and variables

An **atomic operation** is a kind of operation that appears to occur instantaneously to the rest of the tasks of the program. In a concurrent application, you can implement an atomic operation with a critical section to the whole operation using a synchronization mechanism.

An **atomic variable** is a kind of variable that has atomic operations to set and get its value. You can implement an atomic variable using a synchronization mechanism or in a lock-free manner using CAS that doesn't need synchronization.

Problems in concurrent applications:

- Race condition
- DeadLock

There is a **deadlock** in your concurrent application when there are two or more tasks waiting for a shared resource that must be free from another thread that is waiting for another shared resource that must be free by one of the first ones. It happens when four conditions happen simultaneously in the system. They are the **Coffman conditions**, which are as follows:

- **Mutual exclusion**: The resources involved in the deadlock must be nonshareable. Only one task can use the resource at a time.
- **Hold and wait condition**: A task has the mutual exclusion for a resource and it's requesting the mutual exclusion for another resource. While it's waiting, it doesn't release any resources.
- **No pre-emption**: The resources can only be released by the tasks that hold them.
- **Circular wait**: There is a circular waiting where *Task 1* is waiting for a resource that is being held by *Task 2*, which is waiting for a resource being held by *Task 3*, and so on until we have *Task n* that is waiting for a resource being held by *Task 1*.

- Live Lock

A **livelock** occurs when you have two tasks in your system that are always changing their states due to the actions of the other. Consequently, they are in a loop of state changes and unable to continue.

For example, you have two tasks - *Task 1* and *Task 2*, and both need two resources - *Resource 1* and *Resource 2*. Suppose that *Task 1* has a lock on *Resource 1*, and *Task 2* has a lock on *Resource 2*. As they are unable to gain access to the resource they need, they free their resources and begin the cycle again. This situation can continue indefinitely, so the tasks will never end their execution.

- Resource Starvation

Concurrency design patterns

Singnaling

See the following example:

```
public void task1() {
  section1();
  commonObject.notify();
}

public void task2() {
  commonObject.wait();
  section2();
}
```

Rendezvous

```
public void task1() {
  section1_1();
  commonObject1.notify();
  commonObject2.wait();
  section1_2();
}
public void task2() {
  section2_1();
  commonObject2.notify();
  commonObject1.wait();
  section2_2();
}
```

Mutex

```
public void task() {
  preCriticalSection();
  try {
    lockObject.lock() // The critical section begins
    criticalSection();
  } catch (Exception e) {

  } finally {
    lockObject.unlock(); // The critical section ends
     postCriticalSection();
}
```

Multiplex

The **Multiplex design pattern** is a generalization of the Mutex. In this case, a determined number of tasks can execute the critical section at once. It is useful, for example, when you have multiple copies of a resource. The easiest way to implement this design pattern in Java is using the `Semaphore` class initialized to the number of tasks that can execute the critical section at once.

Look at the following example:

```
public void task() {
  preCriticalSection();
  semaphoreObject.acquire();
  criticalSection();
  semaphoreObject.release();
  postCriticalSection();
}
```

Barrier

This design pattern explains how to implement the situation where you need to synchronize some tasks at a common point. None of the tasks can continue with their execution until all the tasks have arrived at the synchronization point. Java Concurrency API provides the `CyclicBarrier` class, which is an implementation of this design pattern.

Look at the following example:

```
public void task() {
  preSyncPoint();
  barrierObject.await();
  postSyncPoint();
}
```

Read-write lock

When you protect access to a shared variable with a lock, only one task can access that variable, independently of the operation you are going to perform on it. Sometimes, you will have variables that you modify a few times but you read many times. In this circumstance, a lock provides poor performance because all the read operations can be made concurrently without any problem. To solve this problem, we can use the read-write lock design pattern. This pattern defines a special kind of lock with two internal locks: one for read operations and another for write operations. The behavior of this lock is as follows:

- If one task is doing a read operation and another task wants to do another read operation, it can do it
- If one task is doing a read operation and another task wants to do a write operation, it's blocked until all the readers finish
- If one task is doing a write operation and another task wants to do an operation (read or write), it's blocked until the writer finishes

The Java Concurrency API includes the class `ReentrantReadWriteLock` that implements this design pattern. If you want to implement this pattern from scratch, you have to be very careful with the priority between read-tasks and write-tasks. If too many read-tasks exist, write-tasks can be waiting too long.

Thread pool

This design pattern tries to remove the overhead introduced by creating a thread per task you want to execute. It's formed by a set of threads and a queue of tasks you want to execute. The set of threads usually has a fixed size. When a thread finishes the execution of a task, it doesn't finish its execution. It looks for another task in the queue. If there is another task, it executes it. If not, the thread waits until a task is inserted in the queue, but it's not destroyed.

The Java Concurrency API includes some classes that implement the `ExecutorService` interface that internally uses a pool of threads.

Tips and tricks for designing concurrent algorithms

- Identifying the correct independent tasks
- Taking scalability into account: (for example, in Java, you can get it with the method **Runtime.getRuntime().availableProcessors**())
- Using thread-safe APIs
- Never assume an execution order
- Using immutable objects when possible
- Avoiding deadlocks by ordering the locks

- Using atomic variables instead of synchronization

15.2 Working with Basic Elements - Threads and Runnable

You can create two kinds of threads in Java:

- Daemon threads
- Non-daemon threads

The difference between them is in how they affect the end of a program. A Java program ends its execution when one of the following circumstances occurs:

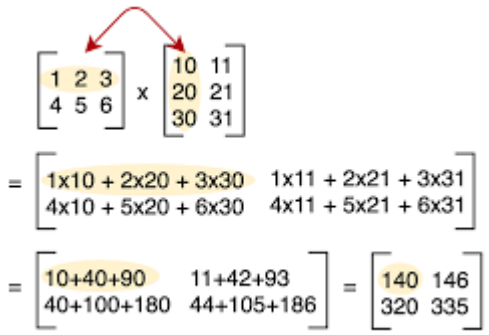
- The program executes the `exit()` method of the `Runtime` class and the user has authorization to execute that method
- All the non-daemon threads of the application have ended its execution, no matter if there are daemon threads running or not

With these characteristics, daemon threads are usually used to execute auxiliary tasks in the applications as garbage collectors or cache managers. You can use the `isDaemon()` method to check if a thread is a daemon thread or not and the `setDaemon()` method to establish a thread as a daemon one. Take into account that you must call this method before the thread starts its execution with the `start()` method.

Finally, threads can pass through different states depending on the situation. All the possible states are defined in the `Thread.States` class and you can use the `getState()` method to get the status of a `Thread`. Obviously, you can change the status of the thread directly. These are the possible statuses of a thread:

- `NEW`: The `Thread` has been created but it hasn't started its execution yet
- `RUNNABLE`: The `Thread` is running in the Java Virtual Machine
- `BLOCKED`: The `Thread` is waiting for a lock
- `WAITING`: The `Thread` is waiting for the action of another thread
- `TIME_WAITING`: The `Thread` is waiting for the action of another thread but has a time limit
- `THREAD`: The `Thread` has finished its execution

Matrix multiplication



Naive solution:

```
public static void multiply(double[][] matrix1,
                            double[][] matrix2,
                            double[][] result){
    int rows1 = matrix1.length;
    int col2 = matrix2[0].length;
    int rows2 = matrix2.length;
    for (int i = 0 ; i < rows1; i++){
        for (int j = 0 ; j < col2; j++){
            int sum = 0 ;
            for (int k = 0 ; k < rows2; k++){
                sum += matrix1[i][k] * matrix2[k][j];
            }
            result[i][j] = sum;
        }
    }
    System.out.println("Result: ");
    for (double[] doubles : result) {
        for (int j = 0; j < result[0].length; j++) {
            System.out.print(doubles[j] + " ");
        }
        System.out.println();
    }
}
public static void main(String[] args) {
    double [][] matrix1 = {{1, 2, 3}, {4, 5, 6}};
    double [][] matrix2 = {{1, 2}, {3, 4}, {5, 6}};
    double [][] result = new double[matrix1.length][matrix2[0].length];
    MatrixMultiplication.multiply(matrix1, matrix2, result);
}
```

Parrallel solution version 1 by element:

```
public class IndividualRunner implements Runnable {
    private final double[][] result;
    private final double[][] matrix1;
    private final double[][] matrix2;
    private final int row;
    private final int col;

    public IndividualRunner(double[][] result,
                            double[][] matrix1,
                            double[][] matrix2,
                            int row,
                            int col) {
        this.result = result;
        this.matrix1 = matrix1;
        this.matrix2 = matrix2;
        this.row = row;
        this.col = col;
    }

    @Override
    public void run() {
        result[row][col] = 0;
        for (int k = 0; k < matrix2.length; k++) {
            result[row][col] += matrix1[row][k]
                                * matrix2[k][col];
        }
    }
}

public class MatrixMultiplicationPerElement {

    public static void multiply(double[][] matrix1,
                                double[][] matrix2,
                                double[][] result,
                                boolean verbose) {

        LocalDateTime start = LocalDateTime.now();
        List<Thread> threads = new ArrayList<>();
        int row1 = matrix1.length;
        int col2 = matrix2[0].length;

        for (int i = 0; i < row1; i++){
            for (int j = 0; j < col2; j++){
                IndividualRunner runner = new IndividualRunner(result,
                                                               matrix1,
                                                               matrix2,
                                                               i, j);
                Thread thread = new Thread(runner);
                thread.start();
                threads.add(thread);
```

```
                if (threads.size() % 10 == 0) {
                    waitForThreads(threads);
                }
            }
        }
    }

    private static void waitForThreads(List<Thread> threads){
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
        }
        threads.clear();
    }
}
```

Parrallel solution version 1 by row:

```java
public class RowRunner implements Runnable {

    private final double[][] result;
    private final double[][] matrix1;
    private final double[][] matrix2;
    private final int row;

    public RowRunner(double[][] result,
                     double[][] matrix1,
                     double[][] matrix2,
                     int i) {
        this.result = result;
        this.matrix1 = matrix1;
        this.matrix2 = matrix2;
        this.row = i;
    }

    @Override
    public void run() {
        int col2 = matrix2[0].length;
        int col1 = matrix1[row].length;

        for (int j = 0; j < col2 ; j++) { // Col 2
            result[row][j] = 0; // Current Row
            for (int k = 0; k < col1; k++) { // Col 1
                result[row][j] += matrix1[row][k] * matrix2[k][j];
            }
        }
    }
}
```

```
public class MatrixMultiplicationPerRow {

    static void multiply(double[][] matrix1,
                         double[][] matrix2,
                         double[][] result,
                         boolean verbose){


        List<Thread> threads = new ArrayList<>();
        int row1 = matrix1.length;
        for (int i = 0; i < row1; i++) {
            RowRunner runner = new RowRunner(result, matrix1, matrix2, i);
            Thread thread = new Thread(runner);
            thread.start();
            threads.add(thread);

            if (threads.size() % 10 == 0) {
                waitForThreads(threads);
            }
        }
    }

    private static void waitForThreads(List<Thread> threads){
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

File search serial:

```
public class SerialFileSearch {

    public static void searchFiles(File file,
                                   String fileName,
                                   Result result){

        File[] contents;
        contents = file.listFiles();
        if ((contents == null) || (contents.length == 0)){
            return;
        }

        for (File content: contents){
            if(content.isDirectory()){
                searchFiles(content, fileName, result);
            } else {
                if (content.getName().equals(fileName)){
                    result.setPath(content.getAbsolutePath());
                    result.setFound(true);
                    System.out.println("Serial file found  " +
                              result.toString());
                    return;
                }
            }

            if (result.isFound()){
                return;
            }
        }
    }

    public static void main(String[] args) {
        File file = new File(System.getProperty("user.dir"));
        Result result = new Result();
        SerialFileSearch.searchFiles(file,
                            "Dummy.class",
                                    result);
    }
}
```

15.3 Thread Executor

Client Server Thread Executor

Server code:

```java
public class RequestTask implements Runnable{

    private final Socket socket;

    public RequestTask(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try (PrintWriter writer = new PrintWriter(
                                socket.getOutputStream(),true);

             BufferedReader reader = new BufferedReader(
                                new InputStreamReader(
                                socket.getInputStream()));)
        {
            String line = reader.readLine();
            System.err.println(line);
            writer.println("Ack !");

        } catch (IOException exception){
            exception.printStackTrace();
        } finally {
            try {
                socket.close();
            } catch (IOException exception) {}
        }
    }
}
```

```
public class ConcurrentServer {
    private static ThreadPoolExecutor executor;
    private static ServerSocket serverSocket;
    private static volatile boolean stopped = false;

    public static void main(String[] args) throws IOException {
        int numCores = Runtime.getRuntime().availableProcessors();
        executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(numCores);
        serverSocket = new ServerSocket(Constants.CONCURRENT_PORT);
        System.out.println("Server initialize success [" + Constants.CONCURRENT_PORT +
"]");
        do {
            try {
                Socket socket = serverSocket.accept();
                RequestTask task = new RequestTask(socket);
                executor.execute(task);
            } catch (IOException exception){
                exception.printStackTrace();
            }
        } while (!stopped);

        try {
            boolean terminate = executor.awaitTermination(1, TimeUnit.DAYS);
            System.out.println("Main server thread ended");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static ThreadPoolExecutor getExecutor() {
        return executor;
    }
}
```

Client Code

```
public class ClientRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < 5; j++) {
                try (Socket socket = new Socket("localhost",
                                        Constants.CONCURRENT_PORT);
                     PrintWriter writer = new PrintWriter(
                                        socket.getOutputStream(),
                                        true);
                     BufferedReader reader = new BufferedReader(
                                        new InputStreamReader(
                                        socket.getInputStream()));){

                    String command = String.format("COMMAND I [%d] J [%d]", i, j);
                    writer.println(command);
                    String output = reader.readLine();
                    System.err.println("OUTPUT: " + output);
                } catch (IOException unknownHostException) {
                    unknownHostException.printStackTrace();
                }
            }
        }
    }
}
```

```java
public class ClientSocket {

    public static void main(String[] args) {
        final int NUM_CLIENTS = 5;
        for (int i = 0; i < NUM_CLIENTS; i++) {
            System.out.println("Number of Simultaneous Clients: " + i);
            Thread[] threads= new Thread[i];
            ClientRunnable client = new ClientRunnable();
            for (int j = 0; j < i; j++) {
                threads[j] = new Thread(client);
                threads[j].start();
            }
            for (int j=0; j<i; j++) {
                try {
                    threads[j].join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Advanced characteristics of executors

Advanced characteristics of executors

An executor is a class that allows programmers to execute concurrent tasks without being worried about the creation and management of threads. Programmers create `Runnable` objects and send them to the executor that creates and manages the necessary threads to execute those tasks. In `Chapter 3`, *Managing Lots of Threads - Executors*, we introduced the basic characteristics of the executor framework:

- How to create an executor and the different options we have when we create one
- How to send a concurrent task to an executor
- How to control the resources used by the executor
- How the executor, internally, uses a pool of threads to optimize the performance of the application

```
int numCores = Runtime.getRuntime().availableProcessors();
executor = (ThreadPoolExecutor) Executors
            .newFixedThreadPool(numCores);
System.out.println("Initialization completed.");
serverSocket = new ServerSocket(Constants.CONCURRENT_PORT);
do {
    try {
        Socket socket = serverSocket.accept();
        RequestTask task = new RequestTask(socket);
        executor.execute(task);
    } catch (IOException exception){
        exception.printStackTrace();
    }
} while (!stopped);

boolean terminated = executor.awaitTermination( timeout: 1, TimeUnit.SECONDS);
System.out.println("Main server thread ended");
```

Callable & Futures:

The Callable interface

The `Callable` interface is very similar to the `Runnable` interface. The main characteristics of this interface are:

- It's a generic interface. It has a single type parameter that corresponds to the return type of the `call()` method.
- It declares the `call()` method. This method will be executed by the executor when it runs the task. It must return an object of the type specified in the declaration.
- The `call()` method can throw any checked exception. You can process the exceptions implementing your own executor and overriding the `afterExecute()` method.

The Future interface

When you send a `Callable` task to an executor, it will return an implementation of the `Future` interface that allows you to control the execution and the status of the task and to get the result. The main characteristics of this interface are:

- You can cancel the execution of the task using the `cancel()` method. This method has a Boolean parameter to specify whether you want to interrupt the task whether it's running or not.

- You can check whether the task has been cancelled (with the `isCancelled()` method) or has finished (with the `isDone()` method).
- You can get the value returned by the task using the `get()` method. There are two variants of this method. The first one doesn't have parameters and returns the value returned by the task if it has finished its execution. If the task hasn't finished its execution, it suspends the execution thread until the tasks finish. The second variant admits two parameters: a period of time and `TimeUnit` of that period. The main difference with the first one is that the thread waits for the period of time passed as a parameter. If the period ends and the task hasn't finished its execution, the method throws a `TimeoutException` exception.

Callable & Future:

```java
class CallableTask implements Callable<Long> {
    int number;

    public CallableTask(int number) {
        this.number = number;
    }

    @Override
    public Long call() throws Exception {
        return factorial(number);
    }

    private long factorial(int n)
            throws InterruptedException {
        long result = 1;
        while (n != 0) {
            result = n * result;
            n = n - 1;
            Thread.sleep(50);
        }
        return result;
    }
}

public class Playground {

    @BeforeEach
    void setUp() {
    }

    @Test
    void parse() throws ExecutionException, InterruptedException {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        System.out.println("Tack Factorial of 10");
        Future<Long> factorial10 = executor.submit(new CallableTask(10));

        System.out.println("Tack Factorial of 15");
        Future<Long> factorial15 = executor.submit(new CallableTask(15));

        System.out.println("Tack Factorial of 20");
        Future<Long> factorial20 = executor.submit(new CallableTask(20));

        System.out.println("Factorial of 10 " + factorial10.get());
        System.out.println("Factorial of 15 " + factorial15.get());
        System.out.println("Factorial of 20 " + factorial20.get());

    }
}
```

Wait for isDone()

```
while(!factorial10.isCancelled()){
    if (factorial10.isDone()) {
        System.out.println("Factorial 10 " + factorial10.get());
        break;
    }
}
while(!factorial15.isCancelled()){
    if (factorial15.isDone()){
        System.out.println("Factorial 15 " + factorial15.get());
        break;
    }
}
while(!factorial20.isCancelled()){
    if (factorial20.isDone()){
        System.out.println("Factorial 20 " + factorial20.get());
        break;
    }
}
```

Important points about Callable and Future

1) Callable is a SAM-type interface, so it can be used in a lambda expression.

2) Callable has just one method `call()` which holds all the code that needs to be executed asynchronously.

3) In the Runnable interface, there was no way to return the result of the computation or throw a checked exception but with Callable, you can both return a value and can throw a checked exception.

4) You can use the `get()` method of Future to retrieve results once the computation is done. You can check if computation is finished or not by using the `isDone()` method.

5) You can cancel the computation by using the `Future.cancel()` method.

6) `get()` is a blocking call and it blocks until the computation is completed.

What's the Future

- `Callable` – `Runnable` on steroids

```
Callable<V> {
  V call() throws Exception;
}
```

- `Future` – result of an asynchronous computation

```
Future<V> {
  V get();
  boolean cancel();
  boolean isCancelled();
  boolean isDone();
}
```

Other methods of interest

In this chapter, we have used some methods of the `AbstractExecutorService` interface (implemented in the `ThreadPoolExecutor` class) and `CompletionService` interfaces (implemented in the `ExecutorCompletionService`) to manage the results of `Callable` tasks. However, there are other versions of the methods we have used and other methods we want to mention here.

Let's discuss the following methods about the `AbstractExecutorService` interface:

- `invokeAll (Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)`: This method returns a list of `Future` objects associated with the list of `Callable` tasks passed as parameters when all the tasks have finished their execution or the timeout specified by the second and third parameters expires.
- `invokeAny (Collection<? Extends Callable<T>> tasks, long timeout, TimeUnit unit)`: This method returns the result of the first task of the list of `Callable` tasks passed as a parameter that finishes their execution without throwing an exception if they finish before the timeout specified by the second and third parameters expires. If the timeout expires, the method throws a `TimeoutException` exception.

Let's discuss the following methods about the `Compl0065tionService` interface:

- The `poll()` method: We have used a version of this method with two parameters, but there is also a version without parameters. From the internal data structures, this version retrieves and removes the `Future` object of the next task that has finished since the last call to the `poll()` or `take()` methods. If no tasks have finished, its execution returns a null value.
- The `take()` method: This method is similar to the previous one, but if no tasks have finished, it sleeps the thread until one task finishes its execution.

Example Building Coordinate Indexing:

Single Callable Calculate on row:

```
public class IndexingTask implements Callable<Map<Integer,
                                    List<Pair<Integer, Integer>>>> {
    private final int[] value;
    private final int row;

    public IndexingTask(int[] value, int row) {
        this.value = value;
        this.row = row;
    }

    @Override
    public Map<Integer, List<Pair<Integer, Integer>>> call() throws Exception {
        Map<Integer, List<Pair<Integer, Integer>>> map = new HashMap<>();
        for (int i = 0; i < value.length; i++) {
            final int temp = i;
            map.merge(value[i], new ArrayList<>(Collections.singleton(Pair.of(row,
temp))),
                    (p1, p2) -> {
                        p1.add(Pair.of(row, temp));
                        return p1;
            });
        }
        return map;
    }
}
```

Aggregate class, take central HashMap and Wait for CompleteService to be done:

```
public class InvertedTask implements Runnable {

private final CompletionService<Map<Integer, List<Pair<Integer, Integer>>>> service;
private final Map<Integer, List<Pair<Integer, Integer>>> finalMap;

public InvertedTask(CompletionService<Map<Integer, List<Pair<Integer, Integer>>>>
service,
                    Map<Integer, List<Pair<Integer, Integer>>> finalMap) {
    this.service = service;
    this.finalMap = finalMap;
}


@Override
public void run() {
    while (!Thread.interrupted()){
        try {
            Map<Integer, List<Pair<Integer, Integer>>> map =
            service.take().get();
            update(map, finalMap);

        } catch (InterruptedException | ExecutionException e) {}
    }

    while (true){
        try {
            Future<Map<Integer, List<Pair<
            Integer, Integer>>>> fut = service.poll();
            if (fut == null)
                break;

            update(fut.get(), finalMap);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}

private void update(Map<Integer, List<Pair<Integer, Integer>>> map,
                    Map<Integer, List<Pair<Integer, Integer>>> finalMap){
    map.forEach((key, val) -> finalMap.merge(key,
        new ArrayList<>(val), (pairs, pairs2) -> {
        pairs.addAll(val);
        return pairs;
        }));
    }
}
```

Main Class:

```java
@Test
void full() {
    LocalDateTime start = LocalDateTime.now();
    int numCores = Runtime.getRuntime().availableProcessors();
    ThreadPoolExecutor executor = (ThreadPoolExecutor)
                    Executors.newFixedThreadPool(numCores);
    CompletionService<Map<Integer, List<Pair<Integer, Integer>>>> service
            = new ExecutorCompletionService<>(executor);
    Map<Integer, List<Pair<Integer, Integer>>> finalMap
            = new ConcurrentHashMap<>();
    int[][] matrix = Data.generate(2500, 2500);
    InvertedTask iTask1 = new InvertedTask(service, finalMap);
    InvertedTask iTask2 = new InvertedTask(service, finalMap);
    Thread thread1 = new Thread(iTask1);
    Thread thread2 = new Thread(iTask2);
    thread1.start();
    thread2.start();
    for (int i = 0; i < matrix.length; i++) {
        IndexingTask task = new IndexingTask(matrix[i], i);
        service.submit(task);
        if (executor.getQueue().size() >= 100) {
            do {
                try {
                    TimeUnit.MILLISECONDS.sleep(50);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } while (executor.getQueue().size() >= 100);
        }
    }
    executor.shutdown();
    try {
        boolean t = executor.awaitTermination(100, TimeUnit.SECONDS);
        thread1.interrupt();
        thread2.interrupt();
        thread1.join();
        thread1.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    /* Performance */
    LocalDateTime end = LocalDateTime.now();
    Duration duration = Duration.between(end, start);
    System.out.println("Duration " + Math.abs(duration.toMillis()) + " ms");
    System.out.println("Size " + finalMap.size());
}
```

15.4 Fork Join Framework

Fork Join Framework

Model

```
Result solve(Problem problem) {
    if (problem is small)
        directly solve problem
    else {
        split problem into independent parts
        fork new subtasks to solve each part
        join all subtasks
        compose result from subresults
    }
}
```

An introduction to the fork/join framework

The executor framework, introduced in Java 5, provides a mechanism to execute concurrent tasks without creating, starting, and finishing threads. This framework uses a pool of threads that executes the tasks you send to the executor, reusing them for multiple tasks. This mechanism provides the following advantages to programmers:

- It's easier to program concurrent applications because you don't have to worry about creating threads.
- It's easier to control the resources used by the executor and your application. You can create an executor that only uses a predefined number of threads. If you send more threads, the executor stores them in a queue until a thread is available.
- Executors reduce the overhead introduced by thread creation by reusing the threads. Internally, it manages a pool of threads that reuses threads to execute multiple tasks.

The divide and conquer algorithm is a very popular design technique. To solve a problem using this technique, you divide it into smaller problems. You repeat the process in a recursive way until the problems you have to solve are small enough to be solved directly. You have to be very careful selecting the base case that is resolved directly. A bad choice of the size of that problem can give you poor performance. This kind of problem can be solved using the executor, but to solve them in a more efficient way, the Java 7 Concurrency API introduced the fork/join framework.

This framework is based on the `ForkJoinPool` class, which is a special kind of executor, two operations, the `fork()` and `join()` methods (and their different variants), and an internal algorithm named the **work-stealing** algorithm. In this chapter, you will learn the basic characteristics, limitations, and components of the fork/join framework in implementing the following three examples:

- The k-means clustering algorithm applied to the clustering of a set of documents
- A data filter algorithm to get the data that meets certain criteria
- The merge sort algorithm to sort big groups of data in an efficient way

As we mentioned before, the fork/join framework must be used to implement solutions to problems based on the divide and conquer technique. You have to divide the original problem into smaller problems until they are small enough to be solved directly. With this framework, you will implement tasks whose main method will be something like this:

```
if ( problem.size() > DEFAULT_SIZE) {
  divideTasks();
  executeTask();
  taskResults=joinTasksResult();
  return taskResults;
} else {
  taskResults=solveBasicProblem();
  return taskResults;
}
```

The most important benefit of this method is that it allows you to divide and execute the child tasks in an efficient way and to get the results of those child tasks to calculate the results of the parent tasks. This functionality is supported by two methods provided by the `ForkJoinTask` class:

- **The fork() method**: This method allows you to send a child task to the fork/join executor
- **The join() method**: This method allows you to wait for the finalization of a child task and returns its result

These methods have different variants, as you will see in the examples. The fork/join framework has another critical feature: the work-stealing algorithm, which determines which tasks are to be executed. When a task is waiting for the finalization of a child task using the `join()` method, the thread that is executing that task takes another task from the pool of tasks that are waiting and starts its execution. In this way, the threads of the fork/join executor are always executing a task by improving the performance of the application.

Limitations of the fork/join framework

As the fork/join framework is used to solve a certain kind of problem, it has some limitations that you have to take into account when you use it to address your problem. These limitations are as follows:

- The basic problems that you're not going to subdivide have to be not very large, but also not very small. According to the Java API documentation, it should have between 100 and 10,000 basic computational steps.
- You should not use blocking I/O operations, such as reading user input or data from a network socket that is waiting until the data is available. Such operations will cause your CPU cores to idle, thereby reducing the level of parallelism, so you will not achieve full performance.
- You can't throw checked exceptions inside a task. You have to include the code to handle them (for example, wrapping into unchecked `RuntimeException`). Unchecked exceptions have special treatment, as you will see in the examples.

Sample Fork Join for get sum for a long list:

```java
class SumAction extends RecursiveTask<Long> {
    private static final int SEQUENTIAL_THRESHOLD = 5;
    private final List<Long> data;

    public SumAction(List<Long> data) {
        this.data = data;
    }

    @Override
    protected Long compute() {
        if (data.size() < SEQUENTIAL_THRESHOLD) {
            long sum = computeSumDirectly();
            System.out.format("Sum of %s: %d\n", data.toString(), sum);
            return sum;
        } else {
            int mid = data.size() / 2;
            SumAction action1 = new SumAction(data.subList(0, mid));
            SumAction action2 = new SumAction(data.subList(mid, data.size()));

            action1.fork();
            return action2.compute() + action1.join();
        }
    }

    private Long computeSumDirectly() {
        long sum = 0;
        for (Long l: data)
            sum += l;
        return sum;
    }
}

public class ForkJoinTest {

    @Test
    void full() {
        Random random = new Random();
        List<Long> data = random
                .longs(10, 1, 10)
                .boxed()
                .collect(toList());

        ForkJoinPool pool = new ForkJoinPool();
        SumAction task = new SumAction(data);
        Long result = pool.invoke(task);
        System.out.println("Total Result " + result);
    }
}
```

Sample Fork Join Search:

Paralled Version

```
public class TaskManager {

    private final Set<RecursiveTask<?>> tasks;
    private final AtomicBoolean cancelled;

    public TaskManager() {
        tasks = ConcurrentHashMap.newKeySet();
        cancelled = new AtomicBoolean(false);
    }

    /**
     *
     * @param sourceTask
     * Cancel all tasks except for the sourceTask
     */
    public void cancel(RecursiveTask<?> sourceTask) {
        if (cancelled.compareAndSet(false, true)) {
            for (RecursiveTask<?> task : tasks) {
                if (task != sourceTask) {
                    if (cancelled.get()) {
                        task.cancel(true);
                    } else {
                        tasks.add(task);
                    }
                }
            }
        }
    }

    public void add(RecursiveTask<?> task) {
        tasks.add(task);
    }

    public void remove(RecursiveTask<?> task) {
        tasks.remove(task);
    }
}
```

Task Find Any

```java
public class IndividualTask extends RecursiveTask<CensusData> {

    private static final long serialVersionUID = -4106884399809219741L;

    private final CensusData[] data;
    private final int start;
    private final int end;
    private final int size;
    private final TaskManager manager;
    private final List<FilterData> filters;

    public IndividualTask(CensusData[] data, int start, int end,
                          TaskManager manager, int size,
                          List<FilterData> filters) {
        this.data = data;
        this.start = start;
        this.end = end;
        this.manager = manager;
        this.size = size;
        this.filters = filters;
    }

    @Override
    protected CensusData compute() {
        if (end - start <= size) {
            for (int i = start; i < end && !Thread.currentThread()
                                                  .isInterrupted(); i++) {
                CensusData censusData = data[i];
                if (Filter.filter(censusData, filters)) {
                    System.out.println("Found: " + i);
                    manager.cancel(this);
                    return censusData;
                }
            }
        } else {
            int mid = (start + end) / 2;
            IndividualTask task1 = new IndividualTask(data, start, mid,
                                                      manager, size, filters);
            IndividualTask task2 = new IndividualTask(data, mid, end,
                                                      manager, size, filters);
            manager.add(task1);
            manager.add(task2);
            manager.remove(this);
            task1.fork(); // Add Task to thread Pool
            task2.fork(); // Add Task to thread Pool

            task1.quietlyJoin();
            task2.quietlyJoin();
            manager.remove(task1);
            manager.remove(task2);
```

```
            try {
                CensusData res = task1.join();
                if (res != null)
                    return res;
                manager.remove(task1);
            } catch (CancellationException ignored) {}

            try {
                CensusData res = task2.join();
                if (res != null)
                    return res;
                manager.remove(task2);
            } catch (CancellationException ignored) {}

        }
        return null;
    }
}
```

List Task:

```
public class ListTask extends RecursiveTask<List<CensusData>> {

    private static final long serialVersionUID = -4106884399809219741L;

    private final CensusData[] data;
    private final int start;
    private final int end;
    private final int size;
    private final List<FilterData> filters;
    private final TaskManager manager;

    public ListTask(CensusData[] data, int start, int end,
                    TaskManager manager, int size, List<FilterData> filters) {
        this.data = data;
        this.start = start;
        this.end = end;
        this.size = size;
        this.filters = filters;
        this.manager = manager;
    }

    @Override
    protected List<CensusData> compute() {
        List<CensusData> ret = new ArrayList<>();
        List<CensusData> tmp;
        if (end - start <= size) {
            for (int i = start; i < end; i++) {
                CensusData censusData = data[i];
                if (Filter.filter(censusData, filters)) {
                    ret.add(censusData);
                }
            }
        } else {
            int mid = (start + end) / 2;
            ListTask task1 = new ListTask(data, start, mid, manager, size, filters);
            ListTask task2 = new ListTask(data, mid, end, manager, size, filters);
            manager.add(task1);
            manager.add(task2);
            manager.remove(this);
            task1.fork();
            task2.fork();
            task2.quietlyJoin();
            task1.quietlyJoin();
            manager.remove(task1);
            manager.remove(task2);

            try {
                tmp = task1.join();
                if (tmp != null)
                    ret.addAll(tmp);
```

```
                manager.remove(task1);
            } catch (CancellationException ignored) {}

            try {
                tmp = task2.join();
                if (tmp != null)
                    ret.addAll(tmp);
                manager.remove(task2);
            } catch (CancellationException ignored) {}
        }

        return ret;
    }
}
```

```
public class ConcurrentSearch {

    public static CensusData findAny(CensusData[] data, List<FilterData> filters, int
size) {

        TaskManager manager = new TaskManager();
        IndividualTask task = new IndividualTask(data, 0,
                                                 data.length,
                                                 manager,
                                                 size,
                                                 filters);
        ForkJoinPool.commonPool().execute(task);
        try {
            CensusData result = task.join();

            if (result != null) {
                System.out.println("Find First Result: "
                        + result.getCitizenship());
            }

            return result;
        } catch (Exception e) {
            System.err.println("findFirst has finished with an error: "
                    + task.getException().getMessage());
        }
        return null;
    }

    public static List<CensusData> findAll(CensusData[] data, List<FilterData>
filters, int size) {
        List<CensusData> results;

        TaskManager manager = new TaskManager();
        ListTask task = new ListTask(data, 0, data.length,
                                     manager, size, filters);
        ForkJoinPool.commonPool().execute(task);

        try {
            results = task.join();
            return results;
        } catch (Exception e) {
            System.err.println("findFirst has finished with an error: "
                        + task.getException().getMessage());
        }
        return null;

    }

}
```

Fork Join Example Search Folder:

```
public class FolderProcessor extends RecursiveTask<List<String>> {
    private static final long serialVersionUID = 1L;
    private final String path;
    private final String extension;

    public FolderProcessor(String path, String extension) {
        this.path = path;
        this.extension = extension;
    }

    @Override
    protected List<String> compute() {
        File file = new File(path);
        File[] contents = file.listFiles();
        List<FolderProcessor> tasks = new ArrayList<>();
        List<String> result = new ArrayList<>();

        if (contents != null) {
            for (File f : contents) {
                if (f.isDirectory()) {
                    FolderProcessor task = new FolderProcessor(
                                                f.getAbsolutePath(),
                                                extension);
                    tasks.add(task);
                    task.fork(); // Fetch to Thread Pool
                } else {
                    if (f.getName().endsWith(extension)) {
                        result.add(f.getName());
                    }
                }
            }
        }
        for (FolderProcessor task : tasks) {
            result.addAll(task.join());
        }
        return result;
    }
}
```

```
public class FolderSearch {

    public static void main(String[] args) {
        ForkJoinPool pool = ForkJoinPool.commonPool();
        FolderProcessor processor = new
        FolderProcessor(System.getProperty("user.dir"), ".java");
        pool.execute(processor);

        do {
            System.out.print("-------------------------------------------------------------
--\n");
            String output = String.format("Parallel [%d] | Active [%d] | Count [%d] |
Steal Count [%d] |",
                                pool.getParallelism(),
                                pool.getActiveThreadCount(),
                                pool.getQueuedTaskCount(),
                                pool.getStealCount());
            System.out.println(output);

            try {
                TimeUnit.MILLISECONDS.sleep(200);
            } catch (InterruptedException ignored) {}
        } while (!processor.isDone());

        pool.shutdown();
        System.out.println("Result found :" + processor.join().size());
    }
}
```

```
------------------------------------------------------------
Parallel [7] | Active [1] | Count [0] | Steal Count [0] |
Result found :88
```

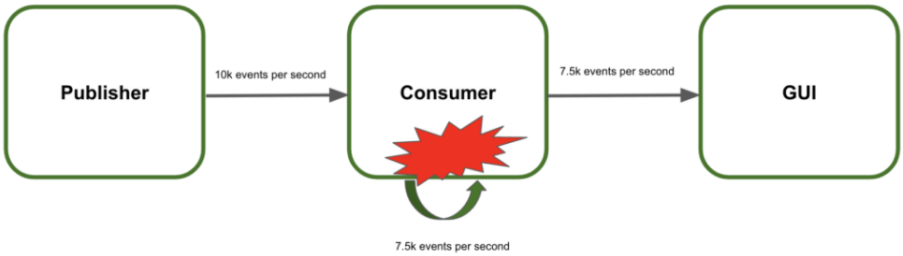
Asynchronous Stream Processing - Reactive Streams

Back Pressure situation:

In Reactive Streams, backpressure also defines how to regulate the transmission of stream elements. In other words, control how many elements the recipient can consume.

Let's use an example to clearly describe what it is:

- The system contains three services: the Publisher, the Consumer, and the Graphical User Interface (GUI)
- The Publisher sends 10000 events per second to the Consumer
- The Consumer processes them and sends the result to the GUI
- The GUI displays the results to the users
- The Consumer can only handle 7500 events per second

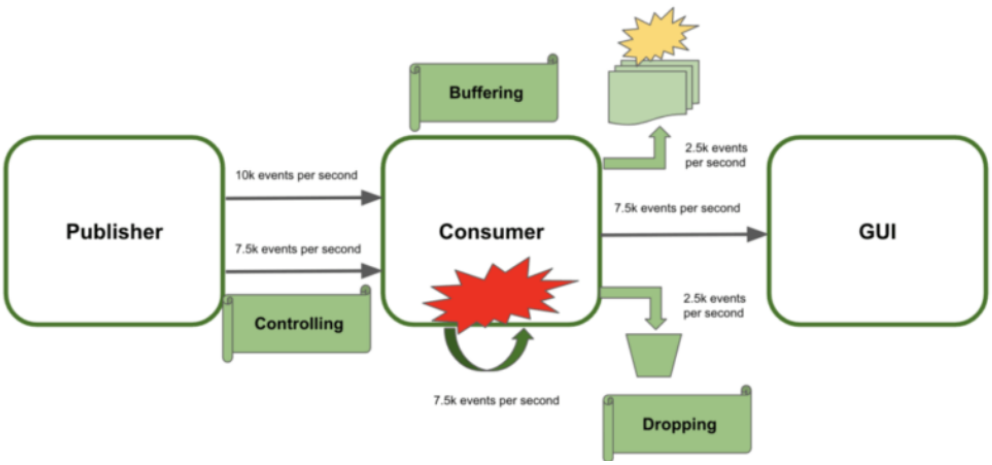


At this speed rate, the consumer cannot manage the events (backpressure). Consequently, the system would collapse and the users would not see the results.

Using Backpressure to Prevent Systemic Failures

The recommendation here would be to apply some sort of backpressure strategy to prevent systemic failures. The objective is to efficiently manage the extra events received:

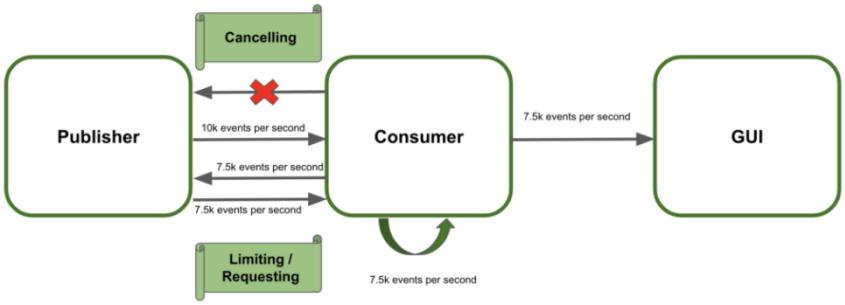
- **Controlling the data stream sent would be the first option**. Basically, the publisher needs to slow down the pace of the events. Therefore, the consumer is not overloaded. Unfortunately, this is not always possible and we would need to find other available options
- **Buffering the extra amount of data is the second choice**. With this approach, the consumer stores temporarily the remaining events until it can process them. The main drawback here is to unbind the buffer causing memory crashing
- **Dropping the extra events losing track of them**. Even this solution is far from ideal, with this technique the system would not collapse



Controlling Backpressure

We'll focus on controlling the events emitted by the publisher. Basically, there are three strategies to follow:

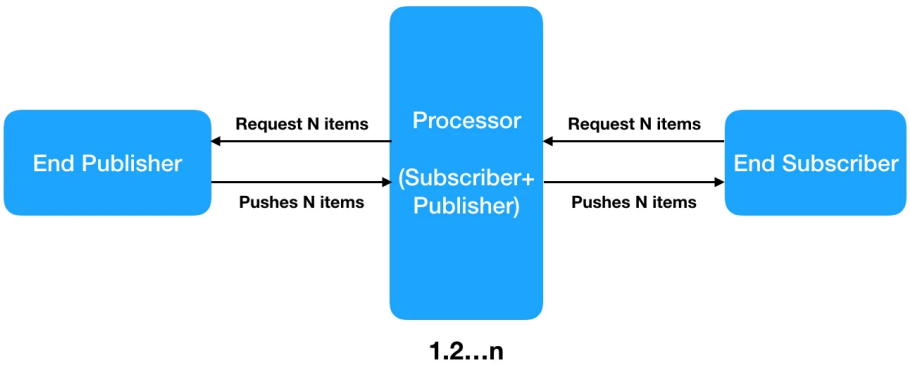
- **Send new events only when the subscriber requests them**. This is a pull strategy to gather elements at the emitter request
- **Limiting the number of events to receive at the client-side**. Working as a limited push strategy the publisher only can send a maximum amount of items to the client at once
- **Canceling the data streaming when the consumer cannot process more events**. In this case, the receiver can abort the transmission at any given time and subscribe to the stream later again

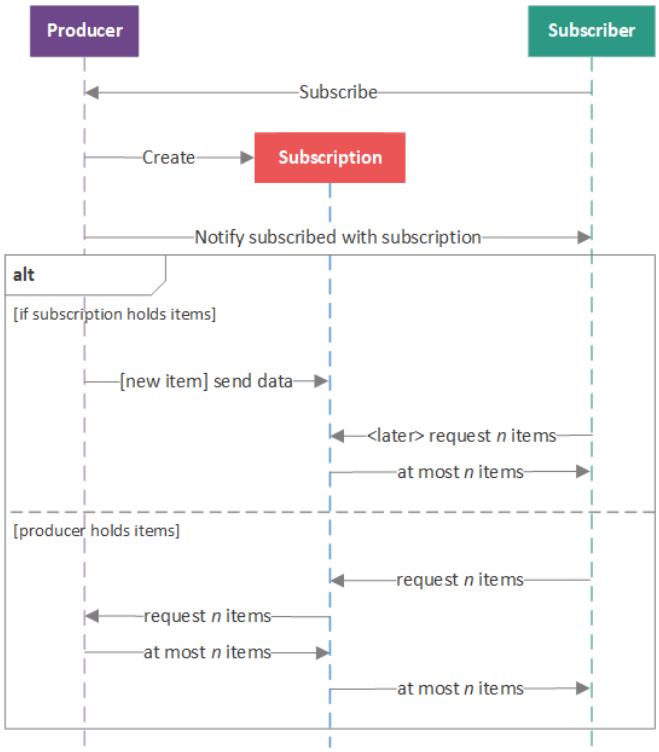


Introduction to reactive streams in Java

In the introduction of this chapter, we explained what reactive streams are, which elements form the standard, and how those elements are implemented in Java:

- **The Flow.Publisher interface**: This interface represents a producer of items.
- **The Flow.Subscriber interface**: This interface represents a consumer of items.
- **The Flow.Subscription interface**: This interface represents the connection between a producer and a consumer. The class that implements it manages the item interchange between the producer and the consumer.





Publisher Interface

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

Subscriber Interface

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

Subscription:

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

Processor:

```
1 public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {}
```

Code sample News Feed System:

News Object

```
@Getter @Setter
@Builder
public class News {
    private String category;
    private String txt;
    private Date date;
}
```

Consumer Object manage its Subscription and categories

```
public class NewsConsumer implements Flow.Subscriber<News> {

    NewsSubscription subscription;
    String name;
    Set<String> interestCategories;

    public NewsConsumer(String name,
                        Set<String> interestCategories) {
        this.name = name;
        this.interestCategories = interestCategories;
    }

    @Override
    public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = (NewsSubscription) subscription;
        this.subscription.setCategories(interestCategories);
        this.subscription.request(1);
        System.out.printf("%s: Consumer [%s] - subscribed ! \n ",
                        Thread.currentThread().getName(), this.name);
    }

    @Override
    public void onNext(News item) {
        System.out.printf("Consumer [%s] get news [%s] \n",
                            this.name, item.getTxt());
        subscription.request(1);
    }

    @Override
    public void onError(Throwable throwable) {
        System.err.println("Error happen" + throwable.getMessage());
    }

    @Override
    public void onComplete() {
        System.out.println("Publisher have closed !");
    }
}
```

Publisher side include of

- **ConsumerData**: meta data class contain info for Subscriber and Subscription
- **NewsSubscription** : implements Flow.Subscription
- **PublisherTask**: implement Runnable, check subcription of ConsumerData and News object.

- **Publisher**: contain **ConcurrentLinkedQueue** List of managed consumer, and an executor for publish new News object.

```java
@Builder
@Getter @Setter
@ToString
public class ConsumerData {
    private NewsSubscription subscription;
    public Flow.Subscriber<? super News> consumer;
}

/* Subscription */
public class NewsSubscription implements Flow.Subscription {
    private boolean isCancel;
    private final AtomicLong request = new AtomicLong(0);
    private Set<String> categories;

    @Override
    publicvoid request(long n) {
        request.addAndGet(n);
    }

    @Override
    public void cancel() {
        isCancel = true;
    }

    public boolean isCancelled() {
        return isCancel;
    }

    public long getRequest() {
        return request.get();
    }

    public void decreaseRequest() {
        request.decrementAndGet();
    }

    public void setCategories(Set<String> categories) {
        this.categories=categories;
    }

    public void setCategories(String category) {
        this.categories.add(category);
    }

    public boolean hasCategories(String category) {
        return this.categories.contains(category);
    }
}

/* Task to push news */
```

```
public class PublisherTask implements Runnable {

    private final ConsumerData consumerData;
    private final News news;

    public PublisherTask(ConsumerData consumerData,
                         News news) {
        this.consumerData = consumerData;
        this.news = news;
    }

    @Override
    public void run() {
        NewsSubscription subscription = consumerData.getSubscription();

        if ( !subscription.isCancelled()
            && subscription.hasCategories(news.getCategory())
            && subscription.getRequest() > 0
        ) {
            consumerData.getConsumer().onNext(news);
            subscription.decreaseRequest();
        }
    }
}

/* News Publisher */
public class NewsPublisher implements Flow.Publisher<News> {
    static int numCores = Runtime.getRuntime().availableProcessors();

    ConcurrentLinkedQueue<ConsumerData> managedConsumers;
    ThreadPoolExecutor executor;

    public NewsPublisher() {
        managedConsumers = new ConcurrentLinkedQueue<>();
        executor = (ThreadPoolExecutor)
        Executors.newFixedThreadPool(numCores);
    }

    @Override
    public void subscribe(Flow.Subscriber<? super News> subscriber) {
        NewsSubscription subscription = new NewsSubscription();
        ConsumerData newConsumer = ConsumerData.builder()
                                               .consumer(subscriber)
                                               .subscription(subscription)
                                               .build();

        managedConsumers.add(newConsumer);
        subscriber.onSubscribe(subscription);
    }

    public void publish(News news) {
        managedConsumers.forEach(consumerData -> {
```

```
            try {
                executor.execute(new PublisherTask(consumerData, news));
            } catch (Exception e) {
                consumerData.getConsumer().onError(e);
            }
        });
    }

    public void shutdown(){
        managedConsumers.forEach(consumerData -> {
            consumerData.getConsumer().onComplete();
        });
        executor.shutdown();
    }
}
```

Main class:

```
public static void main(String[] args) {
    NewsPublisher publisher = new NewsPublisher();
    List<NewsConsumer> consumerList = new ArrayList<>();
    consumerList.add(new NewsConsumer("Duy", Set.of("SPORT", "FUNNY")))
    consumerList.add(new NewsConsumer("Khan", Set.of("SPORT", "FIN")));
    consumerList.add(new NewsConsumer("Sang", Set.of("STOCK")));
    consumerList.forEach(publisher::subscribe);
    publisher.publish(News.builder()
                          .category("SPORT")
                          .date(new Date())
                          .txt("USA [2] vs Brazil [0]")
                          .build());
    publisher.publish(News.builder()
                          .category("FUNNY")
                          .date(new Date())
                          .txt("New comic from Joe")
                          .build());
    publisher.shutdown();
}
```

Diving into Concurrent Data Structures and Synchronization Utilities

Avoid Race conditioning:

To avoid data race conditions, you can:

- Use a non-synchronized data structure and add the synchronization mechanisms by yourself
- Use a data structure provided by the Java Concurrency API that implements the synchronization mechanism internally and is optimized to be used in concurrent applications

Blocking and non-blocking data structures

- **Blocking data structures**: This kind of data structure provides methods to insert and delete data on it that, when the operation cannot be done immediately (for example, you want to take an element and the data structure is empty), the thread that made the call will be blocked until the operation can be done
- **Non-blocking data structures**: This kind of data structure provides methods to insert and delete data on it that, when the operation cannot be done immediately, returns a special value or throws an exception

Concurrent	Blocking
ConcurrentLinkedQueue	LinkedBlockingQueue
ConcurrentLinkedDeque	LinkedBlockingDeque
	ArrayBlockingQueue
	PriorityBlockingQueue
ConcurrentHashMap	

Atomic variables:

Atomic variables

Atomic variables were introduced in Java 1.5 to provide atomic operations over `integer`, `long`, `boolean`, `reference`, and `Array` objects. They provide some methods to increment, decrement, establish the value, return the value, or establish the value if its current value is equal to a predefined one. Atomic variables offer guarantees similar to the `volatile` keyword.

Volatile	Atomic
Volatile only ensures, that at the moment of access of such a variable, the new value will be immediately visible to all other threads and the order of execution ensures, that the code is at the state you would expect it to be. Take the following example:	
i = i + 1;. No matter how you define i, a different Thread reading the value just when the above line is executed might get i, or i + 1, because the operation is not atomically.	Atomics like AtomicInteger ensure, that such operations happen atomically. So the above issue cannot happen, i would either be 1000 or 1001 once both threads are finished.

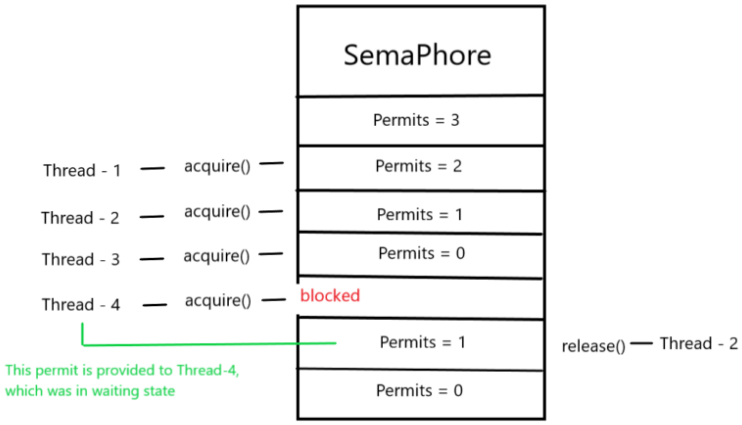
Executor

Type	Description
Single Thread	A thread pool with only one thread. So all the submitted tasks will be executed sequentially. Method : *Executors.newSingleThreadExecutor()*
Cached Thread Pool	A thread pool that creates as many threads it needs to execute the task in parrallel. The old available threads will be **reused** for the new tasks. If a thread is not used during 60 seconds, it will be terminated and removed from the pool.
Fixed Thread Pool	A thread pool with a fixed number of threads. If a thread is not available for the task, the task is put in queue waiting for an other task to ends.
Scheduled Thread Pool	A thread pool made to schedule future task.
Single Thread Scheduled Pool	A thread pool with only one thread to schedule future task

Lock

```
class Task implements Runnable {

    Lock lock = new ReentrantLock();

    @Override
    public void run() {
        try {
            lock.lock(); // Lock Operation
            Random random = new Random();
            System.out.println("Thread ["
                               + Thread.currentThread().getName()
                               + "] Running" );
            TimeUnit.MILLISECONDS.sleep(random.nextInt(1000));
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock(); // Unlock
        }
    }
}

public class LockMain {
    public static void main(String[] args) {
        int numCores = Runtime.getRuntime().availableProcessors();
        ThreadPoolExecutor executor = (ThreadPoolExecutor)
                                      Executors.newFixedThreadPool(numCores);

        for (int i = 0; i < 5; i++) {
            executor.execute(new Task());
        }
        executor.shutdown();
        try {
            boolean terminated = executor.awaitTermination(10, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

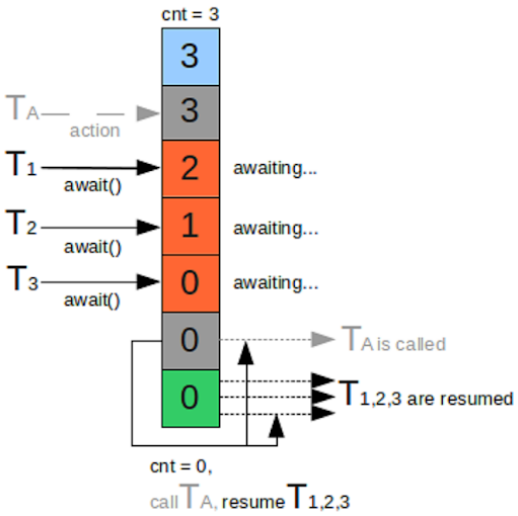
Semaphores


SemaPhore
Permits = 3
Permits = 2
Permits = 1
Permits = 0
Permits = 1
Permits = 0
Thread - 1
acquire()
Thread - 2
acquire()
Thread - 3
acquire()
Thread - 4
acquire()
blocked
release()
Thread - 2
This permit is provided to Thread-4,
which was in waiting state

```java
class SemaphoreTask implements Runnable {

    public Semaphore semaphore;
    public SemaphoreTask(Semaphore semaphore) {
        this.semaphore = semaphore;
    }

    @Override
    public void run() {
        try {
            semaphore.acquire();
            long duration = ThreadLocalRandom.current().nextLong(2000);
            System.out.printf("%s-%s: Working %d ms \n",
                              new Date(),
                              Thread.currentThread().getName(),
                              duration);
            try {
                TimeUnit.MILLISECONDS.sleep(duration);
            } catch (InterruptedException ignored) {}


        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            semaphore.release();
        }
    }
}

public class SemaphoreMain {
    public static void main(String[] args) {

        int numCores = Runtime.getRuntime().availableProcessors();
        ExecutorService executor = Executors.newFixedThreadPool(numCores);
        Semaphore semaphore = new Semaphore(2);
        // Allow two task run simultaneously

        for (int i = 0; i < 10; i++) {
            executor.execute(new SemaphoreTask(semaphore));
        }

        executor.shutdown();
        try {
            boolean terminated = executor.awaitTermination(1, TimeUnit.DAYS);
        } catch (InterruptedException ignored) {}
    }
}
```

CyclicBarrier

Allow all thread await for common point:



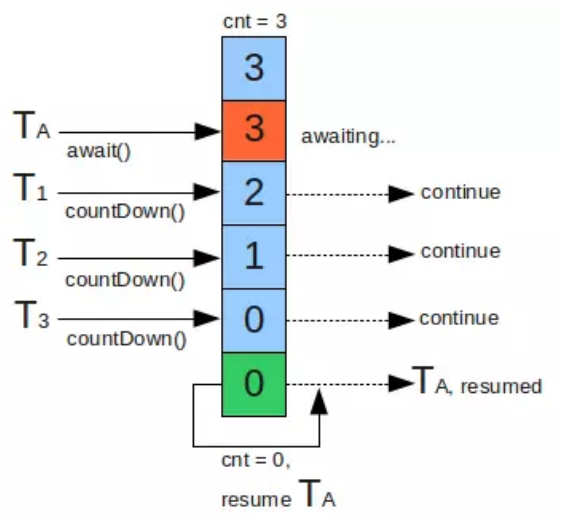
```
class BarrierTask implements Runnable {

    public CyclicBarrier barrier;
    public BarrierTask(CyclicBarrier barrier) {
        this.barrier = barrier;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()
                           + ": Phase 1");

        /* Simulate some task */
        long duration = ThreadLocalRandom.current()
                                         .nextLong(2000);
        System.out.printf("%s: Working %d ms \n",
                Thread.currentThread().getName(),
                duration);
        try {
            TimeUnit.MILLISECONDS.sleep(duration);
        } catch (InterruptedException ignored) {}

        try {
            barrier.await();  // Await for all thread to be done
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName()
                           + ": Phase 2");
    }
}

class FinishTask implements Runnable {
    @Override
    public void run() {
        System.out.println("FinishTask: All the tasks have finished");
    }
}

public class CyclicBarrierMain {
    public static void main(String[] args) {
        int numCores = Runtime.getRuntime().availableProcessors();
        ExecutorService executor = Executors.newFixedThreadPool(numCores);
        CyclicBarrier barrier = new CyclicBarrier(numCores, new FinishTask());

        for (int i=0; i < numCores; i++) {
            executor.execute(new BarrierTask(barrier));
        }
```

```
        executor.shutdown();
        try {
            boolean terminated = executor.awaitTermination(1, TimeUnit.DAYS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
pool-1-thread-4: Phase 1
pool-1-thread-1: Phase 1
pool-1-thread-6: Phase 1
pool-1-thread-5: Phase 1
pool-1-thread-7: Phase 1
pool-1-thread-8: Phase 1
pool-1-thread-2: Phase 1
pool-1-thread-3: Phase 1
pool-1-thread-5: Working 526 ms
pool-1-thread-6: Working 85 ms
pool-1-thread-7: Working 500 ms
pool-1-thread-2: Working 368 ms
pool-1-thread-1: Working 884 ms
pool-1-thread-3: Working 1968 ms
pool-1-thread-4: Working 864 ms
pool-1-thread-8: Working 1327 ms
FinishTask: All the tasks have finished
pool-1-thread-3: Phase 2
pool-1-thread-5: Phase 2
pool-1-thread-4: Phase 2
pool-1-thread-1: Phase 2
pool-1-thread-7: Phase 2
pool-1-thread-2: Phase 2
pool-1-thread-8: Phase 2
pool-1-thread-6: Phase 2

Process finished with exit code 0
```

CountDownLatch

Waiting for counter to zero to complete the task:


cnt = 3
3
3
awaiting...
T_A
await()
T_1
countDown()
2
continue
T_2
countDown()
1
continue
T_3
countDown()
0
continue
0
T_A, resumed
cnt = 0,
resume T_A

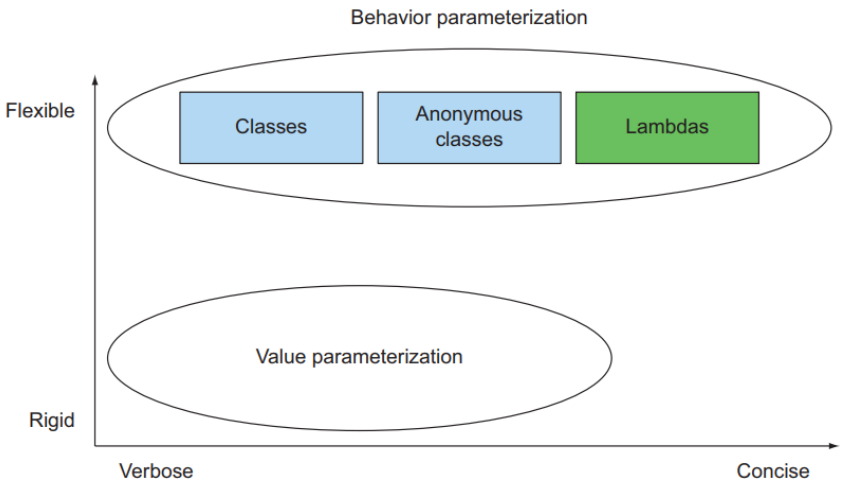
```
class CountDownTask implements Runnable {
    private final CountDownLatch countDownLatch;
    public CountDownTask(CountDownLatch countDownLatch) {
        this.countDownLatch=countDownLatch;
    }

    @SneakyThrows
    @Override
    public void run() {
        Random random = new Random();
        long duration = ThreadLocalRandom.current().nextLong(2000);
        System.out.printf("%s: Working %d ms \n",Thread.currentThread().getName(),
duration);
        TimeUnit.MILLISECONDS.sleep(random.nextInt(3000));

        countDownLatch.countDown();
        System.out.println("Current counter [" + countDownLatch.getCount() + "]");
    }
}

public class CountDownMain {
    public static void main(String[] args) {
        CountDownLatch countDownLatch=new CountDownLatch(10);
        ExecutorService executor= Executors.newCachedThreadPool();
        System.out.println("Main: Launching tasks");

        for (int i = 0; i < 10; i++) {
            executor.execute(new CountDownTask(countDownLatch));
        }
        try {
            countDownLatch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Finish all task");
        executor.shutdown();
    }
}
```

```
Main: Launching tasks
pool-1-thread-2: Working 682 ms
pool-1-thread-4: Working 53 ms
pool-1-thread-5: Working 971 ms
pool-1-thread-1: Working 1998 ms
pool-1-thread-10: Working 1775 ms
pool-1-thread-9: Working 837 ms
pool-1-thread-6: Working 1571 ms
pool-1-thread-8: Working 1478 ms
pool-1-thread-3: Working 1146 ms
pool-1-thread-7: Working 1837 ms
Current counter [9]
Current counter [8]
Current counter [7]
Current counter [6]
Current counter [5]
Current counter [4]
Current counter [3]
Current counter [2]
Current counter [1]
Current counter [0]
Finish all task

Process finished with exit code 0
```

16. Modern Java

16.1 Behaviour parameterization

```java
// Hard Code Using Color
public static List<Apple> filterApplesByColor(List<Apple> inventory, Color color) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
      if (apple.getColor() == color) {
        result.add(apple);
      }
    }
    return result;
  }

// Parameterize Predicate
public static List<Apple> filter(List<Apple> inventory, ApplePredicate p) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple : inventory) {
      if (p.test(apple)) {
        result.add(apple);
      }
    }
    return result;
}

// Using anonymous class
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {
        public boolean test(Apple apple){
            return RED.equals(apple.getColor());
        }
    });

// Using lambda
List<Apple> result = filterApples(inventory, (Apple apple) ->
RED.equals(apple.getColor()));
```



16.2 Lambda

A *lambda expression* can be understood as a concise representation of an anonymous function that can be passed around. It doesn't have a name, but it has a list of parameters, a body, a return type, and also possibly a list of exceptions that can be thrown. That's one big definition; let's break it down:

- *Anonymous*—We say *anonymous* because it doesn't have an explicit name like a method would normally have; less to write and think about!
- *Function*—We say *function* because a lambda isn't associated with a particular class like a method is. But like a method, a lambda has a list of parameters, a body, a return type, and a possible list of exceptions that can be thrown.
- *Passed around*—A lambda expression can be passed as argument to a method or stored in a variable.
- *Concise*—You don't need to write a lot of boilerplate like you do for anonymous classes.

Listing 3.1 Valid lambda expressions in Java 8

```
(String s) -> s.length()
(Apple a) -> a.getWeight() > 150
(int x, int y) -> {
    System.out.println("Result:");
    System.out.println(x + y);
}
```

Takes one parameter of type String and returns an int. It has no return statement as return is implied.

Takes one parameter of type Apple and returns a boolean (whether the apple is heavier than 150 g).

Takes two parameters of type int and returns no value (void return). Its body contains two statements.

This syntax was chosen by the Java language designers because it was well received in other languages, such as C# and Scala. JavaScript has a similar syntax. The basic syntax of a lambda is either (referred to as an *expression-style* lambda)

```
(parameters) -> expression
```

or (note the curly braces for statements, this lambda is often called a *block-style* lambda)

```
(parameters) -> { statements; }
```

Functional interface

Remember the interface `Predicate<T>` you created in chapter 2 so you could parameterize the behavior of the `filter` method? It's a functional interface! Why? Because `Predicate` specifies only one abstract method:

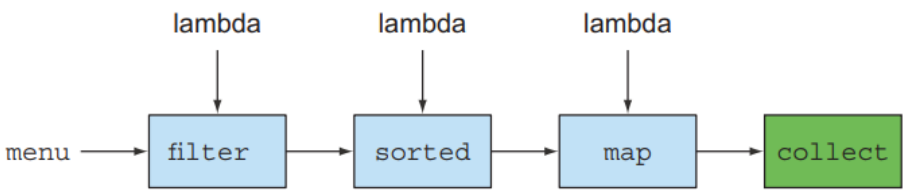
```
public interface Predicate<T> {
    boolean test (T t);
}
```

In a nutshell, a *functional interface* is an interface that specifies exactly one abstract method. You already know several other functional interfaces in the Java API such as `Comparator` and `Runnable`, which we explored in chapter 2:

What are streams?

Streams are an update to the Java API that let you manipulate collections of data in a declarative way (you express a query rather than code an ad hoc implementation for it). For now you can think of them as fancy iterators over a collection of data. In addition, streams can be processed in parallel *transparently*, without you having to write any multithreaded code! We explain in detail in chapter 7 how streams and parallelization work. To see the benefits of using streams, compare the following code to return the

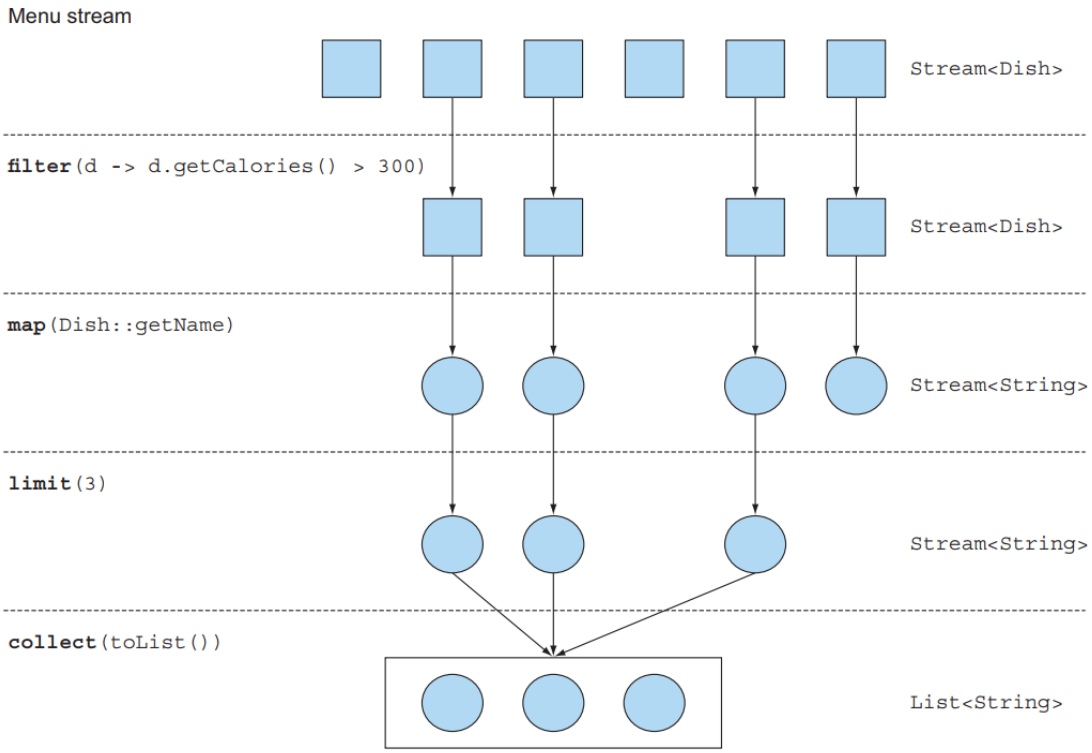
```
// Before Java 8
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish dish: menu) {
    if(dish.getCalories() < 400) {
        lowCaloricDishes.add(dish);
    }
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
    public int compare(Dish dish1, Dish dish2) {
        return Integer.compare(dish1.getCalories(), dish2.getCalories());
    }
});
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish dish: lowCaloricDishes) {
    lowCaloricDishesName.add(dish.getName());
}

// After Java 8
List<String> lowCaloricDishesName = menu.stream()
                                        .filter(d -> d.getCalories() < 400)
                                        .sorted(comparing(Dish::getCalories))
                                        .map(Dish::getName)
                                        .collect(toList());
```



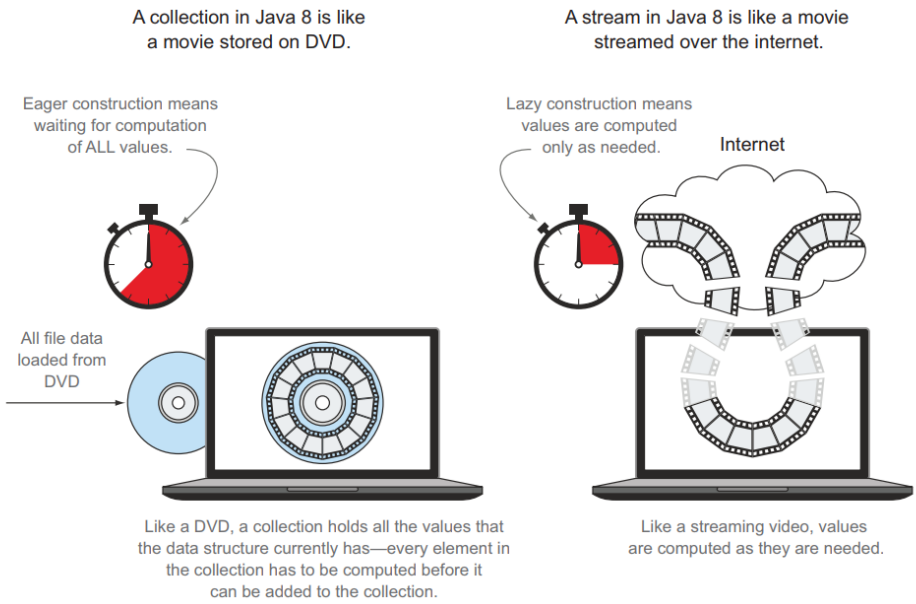
Stream component:

First, what exactly is a *stream*? A short definition is "a sequence of elements from a source that supports data-processing operations." Let's break down this definition step-by-step:

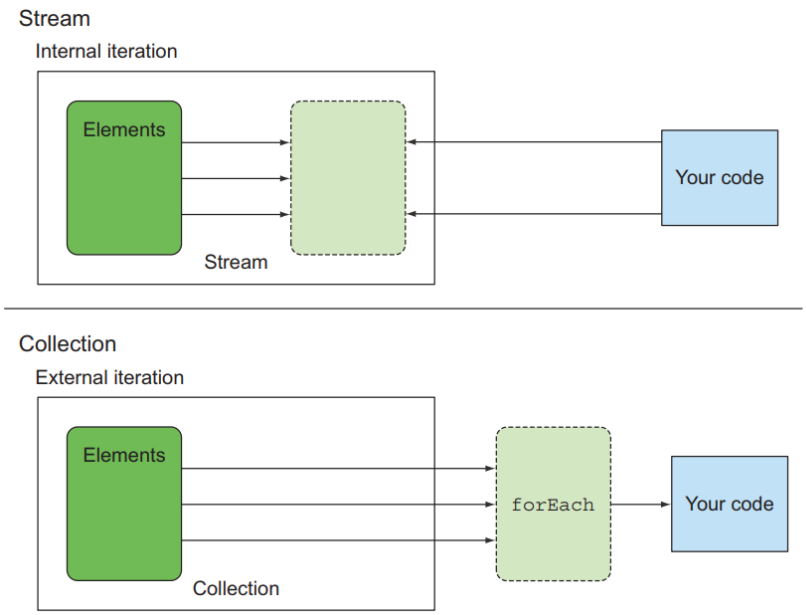
- *Sequence of elements*—Like a collection, a stream provides an interface to a sequenced set of values of a specific element type. Because collections are data structures, they're mostly about storing and accessing elements with specific time/space complexities (for example, an `ArrayList` versus a `LinkedList`). But streams are about expressing computations such as `filter`, `sorted`, and `map`, which you saw earlier. Collections are about data; streams are about computations. We explain this idea in greater detail in the coming sections.
- *Source*—Streams consume from a data-providing source such as collections, arrays, or I/O resources. Note that generating a stream from an ordered collection preserves the ordering. The elements of a stream coming from a list will have the same order as the list.
- *Data-processing operations*—Streams support database-like operations and common operations from functional programming languages to manipulate data, such as `filter`, `map`, `reduce`, `find`, `match`, `sort`, and so on. Stream operations can be executed either sequentially or in parallel.



Streams vs. collections



- Stream is iterate only once.
- Stream is internal interation, collection is external iteration.



To summarize, working with streams in general involves three items:

- A *data source* (such as a collection) to perform a query on
- A chain of *intermediate operations* that form a stream pipeline
- A *terminal operation* that executes the stream pipeline and produces a result

Table 4.1 Intermediate operations

Operation	Type	Return type	Argument of the operation	Function descriptor
`filter`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`map`	Intermediate	`Stream<R>`	`Function<T, R>`	`T -> R`
`limit`	Intermediate	`Stream<T>`		
`sorted`	Intermediate	`Stream<T>`	`Comparator<T>`	`(T, T) -> int`
`distinct`	Intermediate	`Stream<T>`		

Table 4.2 Terminal operations

Operation	Type	Return type	Purpose
`forEach`	Terminal	`void`	Consumes each element from a stream and applies a lambda to each of them.
`count`	Terminal	`long`	Returns the number of elements in a stream.
`collect`	Terminal	(generic)	Reduces the stream to create a collection such as a `List`, a `Map`, or even an `Integer`. See chapter 6 for more detail.

16.4 Stream operation

```java
List<Dish> filteredMenu = specialMenu.stream()
                                     .filter(dish -> dish.getCalories() < 320)
                                     .collect(toList());
filteredMenu.forEach(System.out::println);

System.out.println("Sorted menu sliced with takeWhile():");
List<Dish> slicedMenu1 = specialMenu.stream()
                                    .takeWhile(dish -> dish.getCalories() < 320)
                                    .collect(toList());
slicedMenu1.forEach
(System.out::println);

System.out.println("Sorted menu sliced with dropWhile():");
List<Dish> slicedMenu2 = specialMenu.stream()
                                    .dropWhile(dish -> dish.getCalories() < 320)
                                    .collect(toList());
slicedMenu2.forEach(System.out::println);

// Truncating a stream
List<Dish> dishesLimit3 = menu.stream()
                              .filter(d -> d.getCalories() > 300)
                              .limit(3)
                              .collect(toList());
System.out.println("Truncating a stream:");
dishesLimit3.forEach(System.out::println);

// Skipping elements
List<Dish> dishesSkip2 = menu.stream()
                              .filter(d -> d.getCalories() > 300)
                              .skip(2)
                              .collect(toList());

System.out.println("Skipping elements:");
dishesSkip2.forEach(System.out::println);


// map
List<String> words = Arrays.asList("Hello", "World");
List<Integer> wordLengths = words.stream()
                                 .map(String::length)
                                 .collect(toList());
System.out.println(wordLengths);

// flatMap
words.stream()
     .flatMap((String line) -> Arrays.stream(line.split("")))
     .distinct()
     .forEach(System.out::println);
```

```
  private static boolean isVegetarianFriendlyMenu() {
    return menu.stream().anyMatch(Dish::isVegetarian);
  }

  private static boolean isHealthyMenu() {
    return menu.stream().allMatch(d -> d.getCalories() < 1000);
  }

  private static boolean isHealthyMenu2() {
    return menu.stream().noneMatch(d -> d.getCalories() >= 1000);
  }

  private static Optional<Dish> findVegetarianDish() {
    return menu.stream().filter(Dish::isVegetarian).findAny();
  }


// Reducing
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
System.out.println(sum);

int sum2 = numbers.stream().reduce(0, Integer::sum);
System.out.println(sum2);

int max = numbers.stream().reduce(0, (a, b) -> Integer.max(a, b));
System.out.println(max);

Optional<Integer> min = numbers.stream().reduce(Integer::min);
min.ifPresent(System.out::println);
```

Collecting data with Stream:

```
// Summarize
Comparator<Transaction> comparator = Comparator.comparing(Transaction::getAmount);
Optional<Transaction> mostAmount = list.stream()
                                       .collect(Collectors.maxBy(comparator));

double average = list.stream()
                     .collect(Collectors.averagingDouble(Transaction::getAmount));

String allTradeName = list.stream()
                          .map(Transaction::getTraderName)
                          .collect(Collectors.joining());

Map<String, List<Transaction>> transByCities = list.stream()
                                                   .collect(Collectors.groupingBy(
                                                   Transaction::getOriginLocation));

Map<Boolean, List<Transaction>> transByAmount = list.stream()
                                                    .collect(Collectors
                                                    .partitioningBy(p -> p.getAmount()
> 5000));
```

16.5 Stream Parallel

Best practice of using parallel:

1. Avoid using iterate
2. Associativity: expect results to come without following any order
3. Lambda expressions should be stateless
4. Avoid the modification of the streams' elements
5. Lambda expressions should not emit side effects
6. Only use parallelism when the number of elements is very huge
7. Watch out for boxing. Automatic boxing and unboxing operations can dramatically hurt performance.
8. Some operation such as **findAny** is better for parallel than **findFirst**
9. Checkinf for data structure decomposability

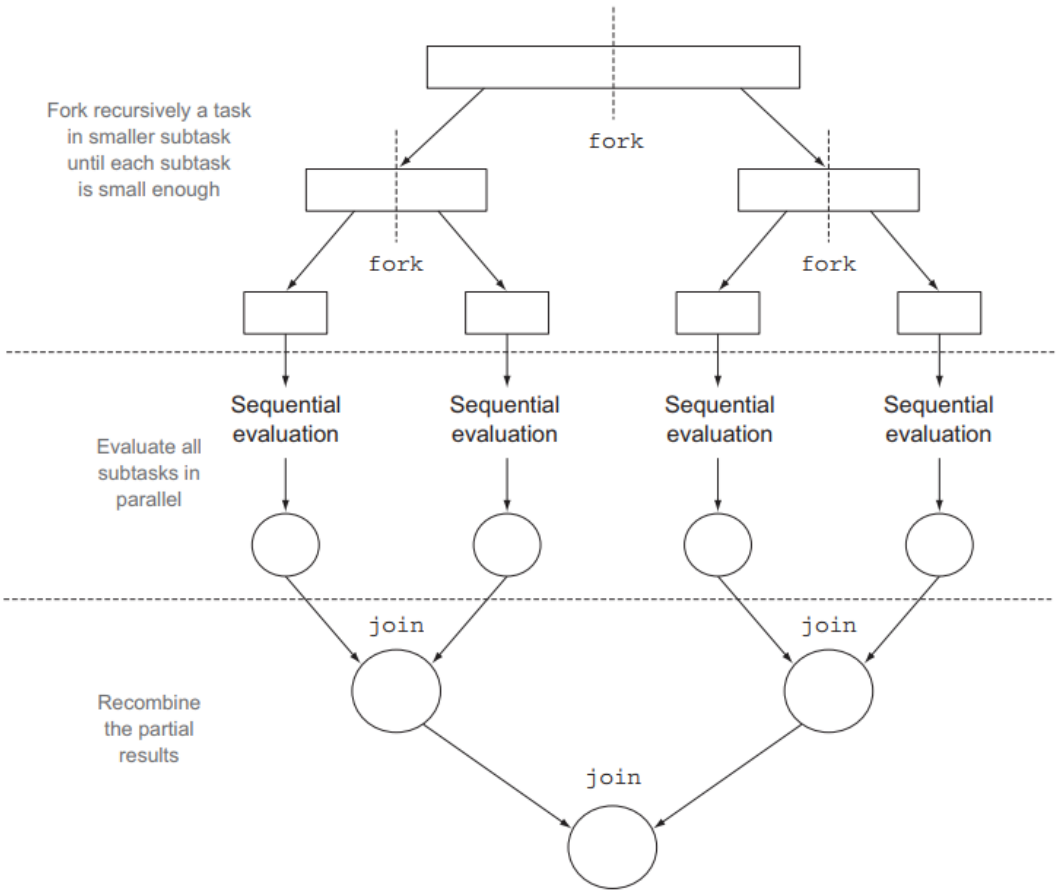
Source	Decomposability
ArrayList	Excellent
LinkedList	Poor
IntStream.range	Excellent
Stream.iterate	Poor
HashSet	Good
TreeSet	Good

```
// Test Value of Stream

public static Function<Integer, Integer> getFunc(String type) {

        switch (type) {
            case "FOR_LOOP" : return integer -> { int sum = 0;
                                        for (int i = 0; i < integer; i++)
                                            sum += i;
                                        return sum; };
            case "SEQUENCE" : return integer -> IntStream.iterate(1, n -> n + 1)
                                                    .limit(integer)
                                                    .reduce(0, Integer::sum);
            case "PARALLEL" : return integer -> IntStream.iterate(1, n -> n + 1)
                                                    .limit(integer)
                                                    .parallel()
                                                    .reduce(0, Integer::sum);
            case "PARALLEL_OPT" : return integer -> IntStream.rangeClosed(1, integer)
                                                        .parallel()
                                                        .reduce(0, Integer::sum);
            default: return null;
        }
    }
```

Result:

```
For Loop start...
[0] :6 ms | [1] :5 ms | [2] :3 ms | [3] :3 ms | [4] :3 ms |
Fastest run :3 ms
-------------
Stream Parallel Optimized start...
[0] :42 ms | [1] :0 ms | [2] :0 ms | [3] :0 ms | [4] :0 ms |
Fastest run :0 ms
-------------
Stream Parallel start...
[0] :203 ms | [1] :128 ms | [2] :135 ms | [3] :140 ms | [4] :123 ms |
Fastest run :123 ms
-------------
Stream Sequence start...
[0] :94 ms | [1] :132 ms | [2] :80 ms | [3] :80 ms | [4] :64 ms |
Fastest run :64 ms
-------------

Process finished with exit code 0
```

Fork/ Join framework



- Invoking the `join` method on a task blocks the caller until the result produced by that task is ready. For this reason, it's necessary to call it after the computation of both subtasks has been started. Otherwise, you'll end up with a slower and more complex version of your original sequential algorithm because every subtask will have to wait for the other one to complete before starting.
- The `invoke` method of a `ForkJoinPool` shouldn't be used from within a `RecursiveTask`. Instead, you should always call the methods `compute` or `fork` directly; only sequential code should use `invoke` to begin parallel computation.
- Calling the `fork` method on a subtask is the way to schedule it on the `ForkJoinPool`. It might seem natural to invoke it on both the left and right subtasks, but this is less efficient than directly calling `compute` on one of them. Doing this allows you to reuse the same thread for one of the two subtasks and avoid the overhead caused by the unnecessary allocation of a further task on the pool.

16.6 Effective Programming with streams and lambdas

16.6.1 Collection API Enhancement

```
public static void main(String[] args) throws NoSuchAlgorithmException {
    // Collection factory: create immutable collection
    List<String> actors = List.of("Keanu", "Jessica");
    Set<String> friends = Set.of("Raphael", "Olivia", "Thibaut");
    Map<String, Integer> ageOfFriends = Map.of("Raphael", 30, "Olivia", 25, "Thibaut",
26);

    // Working with List and Set
    List<String> data = new java.util.ArrayList<>(List.of("Keanu", "Jessica"));
    data.removeIf(s -> s.length() > 10); // remove if len > 10
    data.replaceAll(code -> code.toUpperCase(Locale.ROOT)); // Upper case all
    System.out.println(data);

    // getOrDefault
    Map<String, String> favouriteMovies = Map.ofEntries(entry("Raphael", "Star Wars"),
                                                        entry("Olivia", "James
Bond"));
    System.out.println(favouriteMovies.getOrDefault("Olivia", "Matrix")); // James
Bond
    System.out.println(favouriteMovies.getOrDefault("Thibaut", "Matrix")); // Matrix
}
```

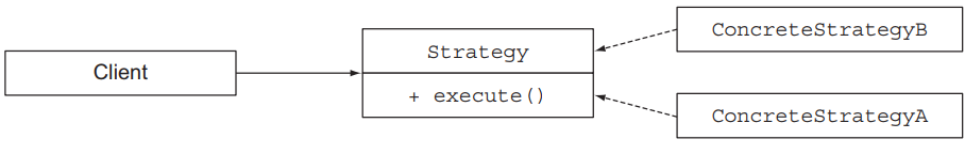
16.6.1 Refactoring, testing, and debugging

Conditional execution

```
// bad code
if (logger.isLoggable(Log.FINER)) {
     logger.finer("Problem: " + generateDiagnostic());
}

// good code
logger.log(Level.FINER, "Problem: " + generateDiagnostic());
```

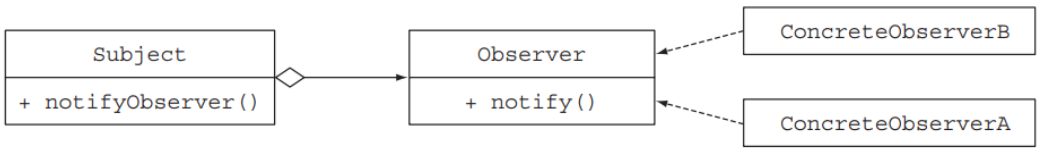
Refactoring design pattern

Strategy:



```
public class Strategy {

    // Define Functional interface
    interface ValidationStrategy {
        boolean execute(String s);
    }

    static private class IsAllLowerCase
        implements ValidationStrategy {

        @Override
        public boolean execute(String s) {
            return s.matches("[a-z]+");
        }

        @Override
        public String toString() { return "IsAllLowerCase"; }
    }

    static private class IsNumeric
        implements ValidationStrategy {

        @Override
        public boolean execute(String s) {
            return s.matches("\\d+");
        }

        @Override
        public String toString() { return "IsNumeric"; }
    }

    static void validate(ValidationStrategy strategy, String name) {
        String output = String.format("Validate %s using strategy %s -> result %b",
                                      name,
                                      strategy.toString(),
                                      strategy.execute(name));
        System.out.println(output);
    }

    public static void main(String[] args) {
        validate(new IsNumeric(), "1231"); // true
        validate(new IsNumeric(), "duyntc"); // false

        validate(new IsAllLowerCase(), "1231"); // false
        validate(new IsAllLowerCase(), "duyntc"); // true
    }
}
```

Observer:



Two interface for **Observer** and **Subject**:

```
interface Subject {
    void registerObserver(Observer o);
    void notifyObservers(String tweet);
}

interface Observer {
    void inform(String tweet);
}
```

Implementation of Observer:

```
/* List of Observer implementation */
    static private class NYTimes implements Observer {
        @Override
        public void inform(String tweet) {
            if (tweet != null && tweet.contains("money"))
                System.out.println("Breaking news in NY!" + tweet);
        }
    }

    static private class Guardian implements Observer {
        @Override
        public void inform(String tweet) {
            if (tweet != null && tweet.contains("queen"))
                System.out.println("Yet another news in London... " + tweet);
        }
    }

    static private class LeMonde implements Observer {
        @Override
        public void inform(String tweet) {
            if (tweet != null && tweet.contains("wine"))
                System.out.println("Today cheese, wine and news! " + tweet);
        }
    }
```

Subject implementation:

```
/* News Feed Implementation */
    static private class NewsFeed implements Subject {
        private final List<Observer> observers = new ArrayList<>();
        @Override
        public void registerObserver(Observer o) {
            observers.add(o);
        }

        @Override
        public void notifyObservers(String tweet) {
            observers.forEach(observer -> observer.inform(tweet));
        }
    }
```

16.7 Optional

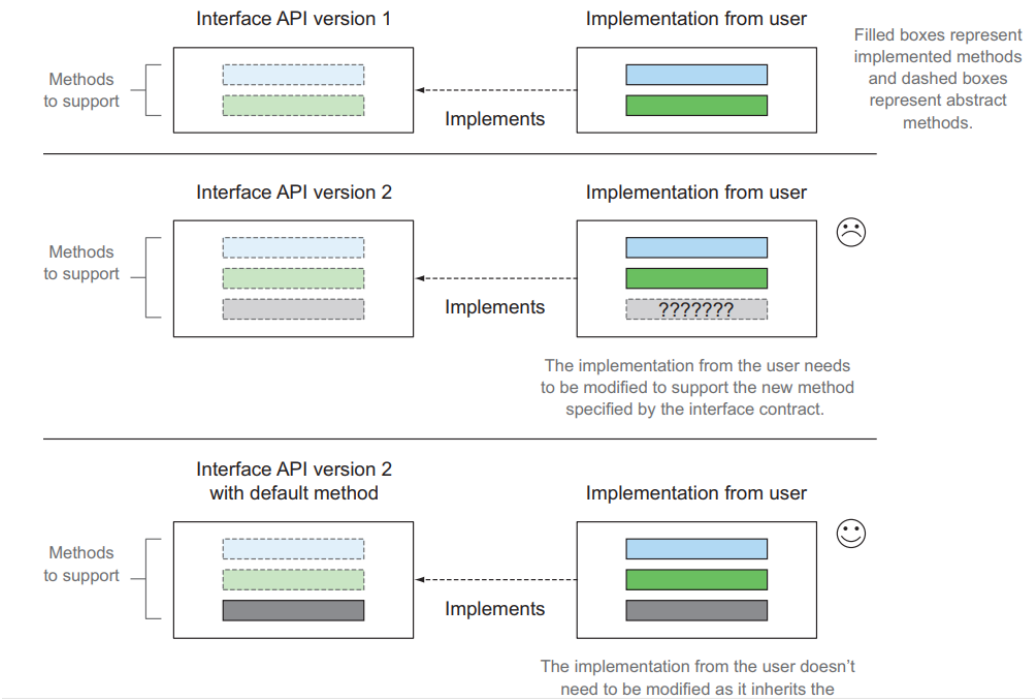
Without Optional

```
public class Person {
    private Car car;
    public Car getCar() {
        return car;
    }
}

public class Car {
    private Insurance insurance;
    public Insurance getInsurance() {
        return insurance;
    }
}

public class Insurance {
    private String name;
    public String getName() {
        return name;
    }
}

// Can potentially genergate Nullpointer Exception
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName(); // <- Null Pointer here
}


// Nested Null check -> Ugly Code
public String getCarInsuranceName(Person person) {
    if (person != null) {
        Car car = person.getCar();
        if (car != null) {
            Insurance insurance = car.getInsurance();
            if (insurance != null) {
                return insurance.getName();
            }
        }
    }
    return "Unknown";
}
```

With Optional

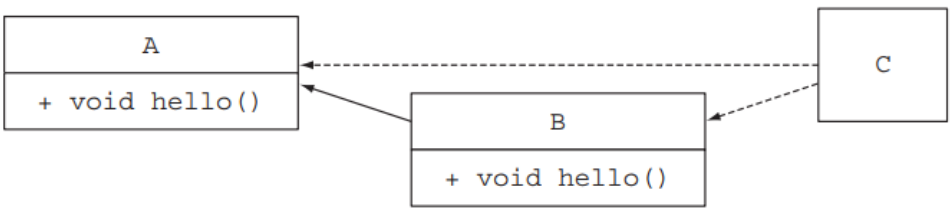
```java
public String getCarInsuranceName(Optional<Person> person, int minAge) {
    return person.filter(p -> p.getAge() >= minAge)
     .flatMap(Person::getCar)
     .flatMap(Car::getInsurance)
     .map(Insurance::getName)
     .orElse("Unknown");
  }
```

16.8 Default Method

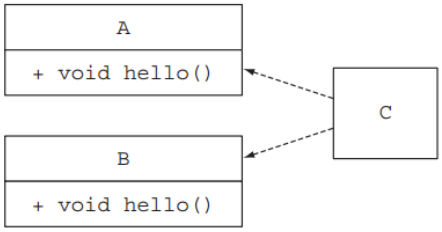


<u>Some rules about default methods:</u>

- **Most specific default-providing interface wins**: this example hello() from B is print



- **Conflicts and explicit disambiguation**: this case require explicit override



16.9 Java Module System

Modularity limitations

Unfortunately, the built-in support in Java to help produce modular software projects was somewhat limited before Java 9. Java has had three levels at which code was grouped: classes, packages, and JARs. For classes, Java has always had support for access modifiers and encapsulation. There was little encapsulation at the package and JAR levels, however.

LIMITED VISIBILITY CONTROL

As discussed in the previous section, Java provides access modifiers to support information hiding. These modifiers are public, protected, package-level, and private visibility. But what about controlling visibility *between* packages? Most applications have several packages defined to group various classes, but packages have limited support for visibility control. If you want classes and interfaces from one package to be visible to another package, you have to declare them as public. As a consequence, these classes and interfaces are accessible to everyone else as well. A typical occurrence of this problem is when you see companion packages with names that include the string `"impl"` to provide default implementations. In this case, because the code inside that package was defined as public, you have no way to prevent users from using these internal implementations. As a result, it becomes difficult to evolve your code without making breaking changes, because what you thought was for internal use only was used by a programmer temporarily to get something working and then frozen into the system. Worse, this situation is bad from a security point of view because you potentially increase the attack surface as more code is exposed to the risk of tampering.

Class Path:

Unfortunately, the combination of the class path and JARs has several downsides.

First, the class path has no notion of versioning for the same class. You can't, for example, specify that the class JSONParser from a parsing library should belong to version 1.0 or version 2.0, so you can't predict what will happen if the same library with two different versions is available on the class path. This situation is common in large applications, as you may have different versions of the same libraries used by different components of your application.

Second, the class path doesn't support explicit dependencies; all the classes inside different JARs are merged into one bag of classes on the class path. In other words, the class path doesn't let you declare explicitly that one JAR depends on a set of classes contained inside another JAR. This situation makes it difficult to reason about the class path and to ask questions such as:

- Is anything missing?
- Are there any conflicts?

Monolithic JDK

Monolithic JDK

The *Java Development Kit* (JDK) is a collection of tools that lets you work with and run Java programs. Perhaps the most important tools you're familiar with are `javac` to compile Java programs and `java` to load and run a Java application, along with the JDK library, which provides runtime support including input/output, collections, and streams. The first version was released in 1996. It's important to understand that like any software, the JDK has grown and increased considerably in size. Many technologies were added and later deprecated. CORBA is a good example. It doesn't matter whether you're using CORBA in your application or not; its classes are shipped with the JDK. This situation becomes problematic especially in applications that run on mobile or in the cloud and typically don't need all the parts available in the JDK library.

Java Module system:

Java modules: the big picture

Java 9 provides a new unit of Java program structure: the *module*. A module is introduced with a new keyword[2] `module`, followed by its name and its body. Such a *module descriptor*[3] lives in a special file: `module-info.java`, which is compiled to `module-info.class`. The body of a module descriptor consists of clauses, of which the two most important are `requires` and `exports`. The former clause specifies what other modules your modules need to run, and `exports` specifies everything that your module wants to be visible for other modules to use. You learn about these clauses in more detail in later sections.

A module descriptor describes and encapsulates one or more packages (and typically lives in the same folder as these packages), but in simple use cases, it exports (makes visible) only one of these packages.

The core structure of a Java module descriptor is shown in figure 14.2.

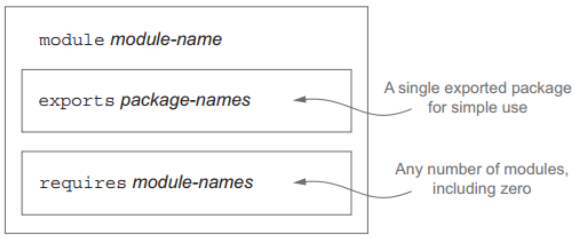


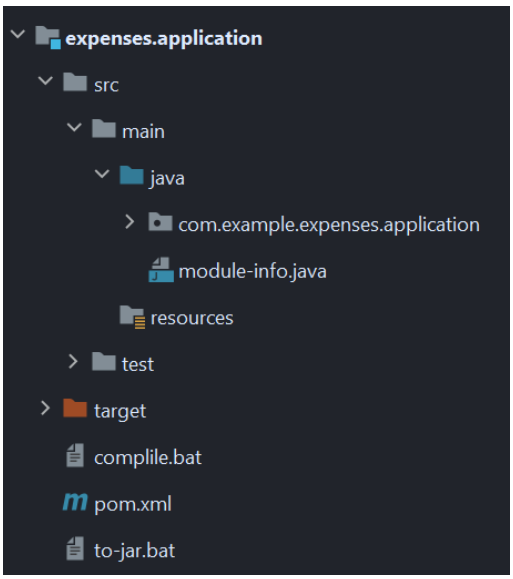
Figure 14.2 Core structure of a Java module descriptor (`module-info.java`)

It is helpful to think of the `exports` and `requires` parts of a module as being respectively like the lugs (or tabs) and holes of a jigsaw puzzle (which is perhaps where the working name Project Jigsaw originated). Figure 14.3 shows an example with several modules.

Java Module System basics

Structure sample project

expense.application module:




```
</parent>
<modelVersion>4.0.0</modelVersion>

<artifactId>expenses.application</artifactId>

<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>org.example</groupId>
        <artifactId>expenses.readers</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
</dependencies>
```

expense.readers module:

```
module expenses.readers {
    exports com.example.expenses.readers;
}
```

contain module:

```xml
<modules>
    <module>expenses.application</module>
    <module>expenses.readers</module>
</modules>

<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>11</source>
                    <target>11</target>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
```

Some module declarations and clauses:

- requires
- export
- requires transitive
- open: allow module reflective

16.10 Java Async Module

Replace Sequential sync call with async:

```
// Sync code
 private static void sequential(int x) {
    int y = f(x);
    int z = g(x);
    System.out.println(y + z);
}

// Async code
  public static Future<Integer> ff(int x) {
    return new CompletableFuture<Integer>()
                .completeAsync(() -> Integer.valueOf(x * 2));
  }

  public static Future<Integer> gf(int x) {
    return new CompletableFuture<Integer>()
              .completeAsync(() -> Integer.valueOf(x + 1));
  }

  private static void futureBased(int x) throws InterruptedException,
ExecutionException {
    Future<Integer> y = ff(x);
    Future<Integer> z = gf(x);
    System.out.println(y.get() + z.get());
  }
```

CompletableFuture and combinators for concurrency

```
public static void main(String[] args) throws ExecutionException,
                                               InterruptedException {
    ExecutorService executorService = Executors.newFixedThreadPool(10);
    int x = 1337;
    CompletableFuture<Integer> a = new CompletableFuture<>();
    CompletableFuture<Integer> b = new CompletableFuture<>();
    CompletableFuture<Integer> c = a.thenCombine(b, (y, z)-> y + z);
    executorService.submit(() -> a.complete(f(x)));
    executorService.submit(() -> b.complete(g(x)));
    System.out.println(c.get());
    executorService.shutdown();
}
```

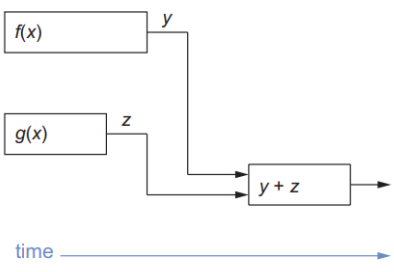


Figure 15.8 Timing diagram showing three computations: `f(x)`, `g(x)` and adding their results

16.11 CompletableFuture composable async

Submit a Callable to the ExecutorService.

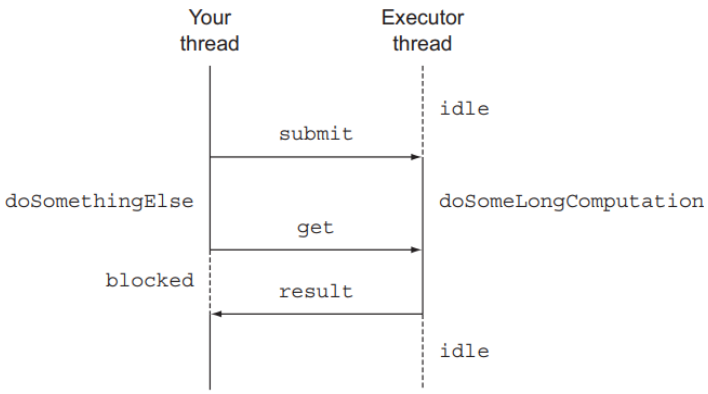
Create an ExecutorService allowing you to submit tasks to a thread pool.

```
ExecutorService executor = Executors.newCachedThreadPool();
Future<Double> future = executor.submit(new Callable<Double>() {
        public Double call() {
            return doSomeLongComputation();
        }});
doSomethingElse();
try {
    Double result = future.get(1, TimeUnit.SECONDS);
} catch (ExecutionException ee) {
    // the computation threw an exception
} catch (InterruptedException ie) {
    // the current thread was interrupted while waiting
} catch (TimeoutException te) {
    // the timeout expired before the Future completion
}
```

Do something else while the asynchronous operation is progressing.

Retrieve the result of the asynchronous operation, blocking if it isn't available yet but waiting for 1 second at most before timing out.

Execute a long operation asynchronously in a separate thread.



Async code:

```
  public Future<Double> getPrice(String product) {

    CompletableFuture<Double> futurePrice = new CompletableFuture<>();

    new Thread(() -> {
      try {
        double price = calculatePrice(product); // Long Process
        futurePrice.complete(price);
      } catch (Exception ex) {
        futurePrice.completeExceptionally(ex);
      }
    }).start();

    return futurePrice;

  }

 public static void main(String[] args) {
    AsyncShop shop = new AsyncShop("BestShop");
    Future<Double> futurePrice = shop.getPrice("myPhone");

    try {
      System.out.println("Price is " + futurePrice.get());
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
```

Using **supplyAsync**: The above code can be replace with

```
public Future<Double> getPriceAsync(String product) {
    return CompletableFuture.supplyAsync(() -> calculatePrice(product));
}
```

Making code Async:

```
// Sequence
public List<String> findPricesSequential(String product) {
    return shops.stream()
                .map(shop -> shop.getPrice(product))
                .map(Quote::parse)
                .map(Discount::applyDiscount)
                .collect(Collectors.toList());
}

// Parallel
public List<String> findPricesParallel(String product) {
    return shops.parallelStream()
                .map(shop -> shop.getPrice(product))
                .map(Quote::parse)
                .map(Discount::applyDiscount)
                .collect(Collectors.toList());
}

// Async
public List<String> findPricesFuture(String product) {
    List<CompletableFuture<String>> priceFutures =
            shops.stream()
                 .map(shop -> CompletableFuture.supplyAsync(
                         () -> shop.getName() + " price is " +
                                 shop.getPrice(product)))
                 .collect(Collectors.toList());
    return priceFutures.stream()
                       .map(CompletableFuture::join)
                       .collect(toList());
}


Time take 1111 ms Future
Time take 2040 ms Parallel
Time take 10141 ms Sequence
```

The CompletableFutures version has an advantage: by contrast with the parallel Streams API, it allows you to specify a different Executor to submit tasks to. You can configure this Executor, and size its thread pool, in a way that better fits the requirements of your application.

Sizing thread pools

In the great book *Java Concurrency in Practice* (Addison-Wesley, 2006; http://jcip.net), Brian Goetz and his co-authors give some advice on finding the optimal size for a thread pool. This advice is important because if the number of threads in the pool is too big, the threads end up competing for scarce CPU and memory resources, wasting their time performing context switching. Conversely, if this number is too small (as it likely is in your application), some of the cores of the CPU will remain underused. Goetz suggests that you can calculate the right pool size to approximate a desired CPU use rate with the following formula:

$$N^{threads} = N^{CPU} * U^{CPU} * (1 + W/C)$$

In this formula, NCPU is the number of cores, available through `Runtime.getRuntime().availableProcessors()`

- U^{CPU} is the target CPU use (between 0 and 1).
- W/C is the ratio of wait time to compute time.

Customize thread executor:

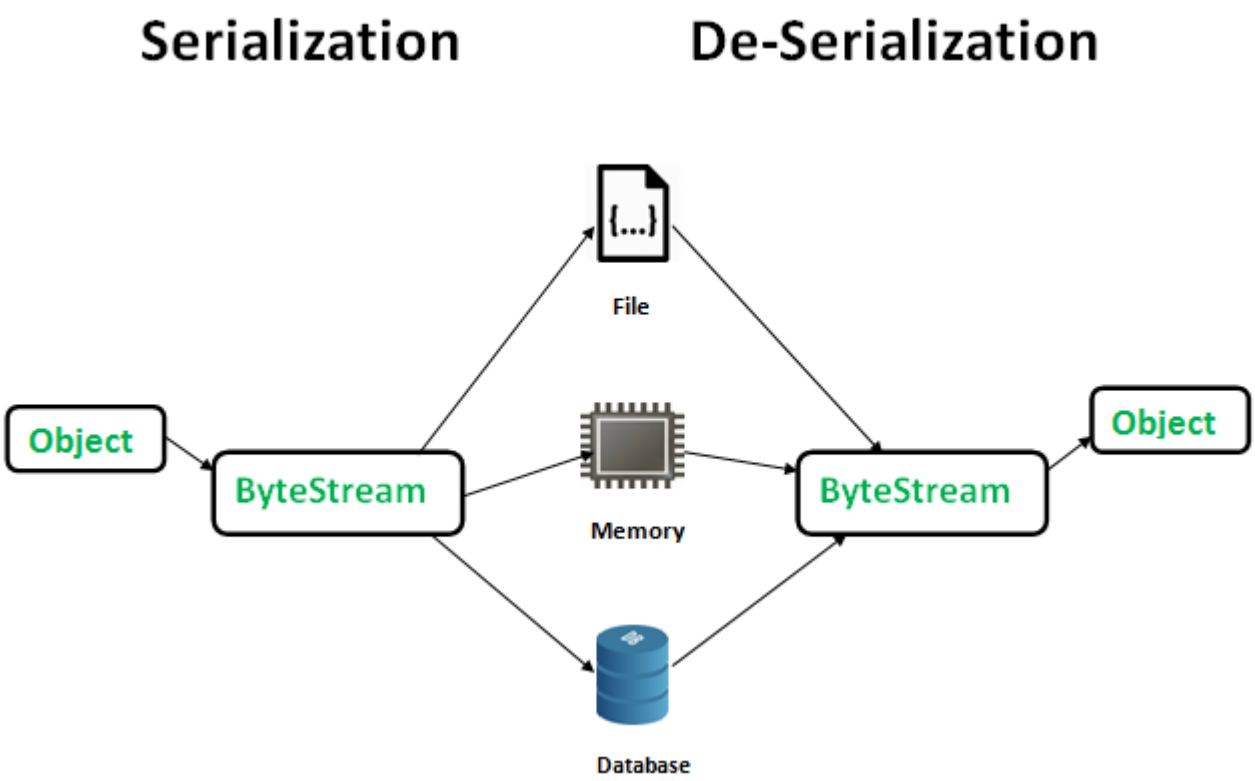
```
private final ThreadFactory setDaemonFactory = (Runnable r) -> {
        Thread t = new Thread(r);
        t.setDaemon(true);
        return t;
    };

private final Executor optExecutor =
Executors.newFixedThreadPool(Math.min(shops.size(), 100),
                                                    setDaemonFactory);
```

Performance boost:

```
public List<String> findPricesOpt(String product) {
    List<CompletableFuture<String>> priceFutures =
            shops.stream()
                 .map(shop -> CompletableFuture.supplyAsync(
                         () -> shop.getName() + " price is " +
                                 shop.getPrice(product), optExecutor)) // Customoze
executor
                 .collect(Collectors.toList());
    return priceFutures.stream()
                       .map(CompletableFuture::join)
                       .collect(toList());
}


Time take 1078 ms Future
Time take 2016 ms Parallel
Time take 14105 ms Sequence
Time take 1004 ms Optimize
```

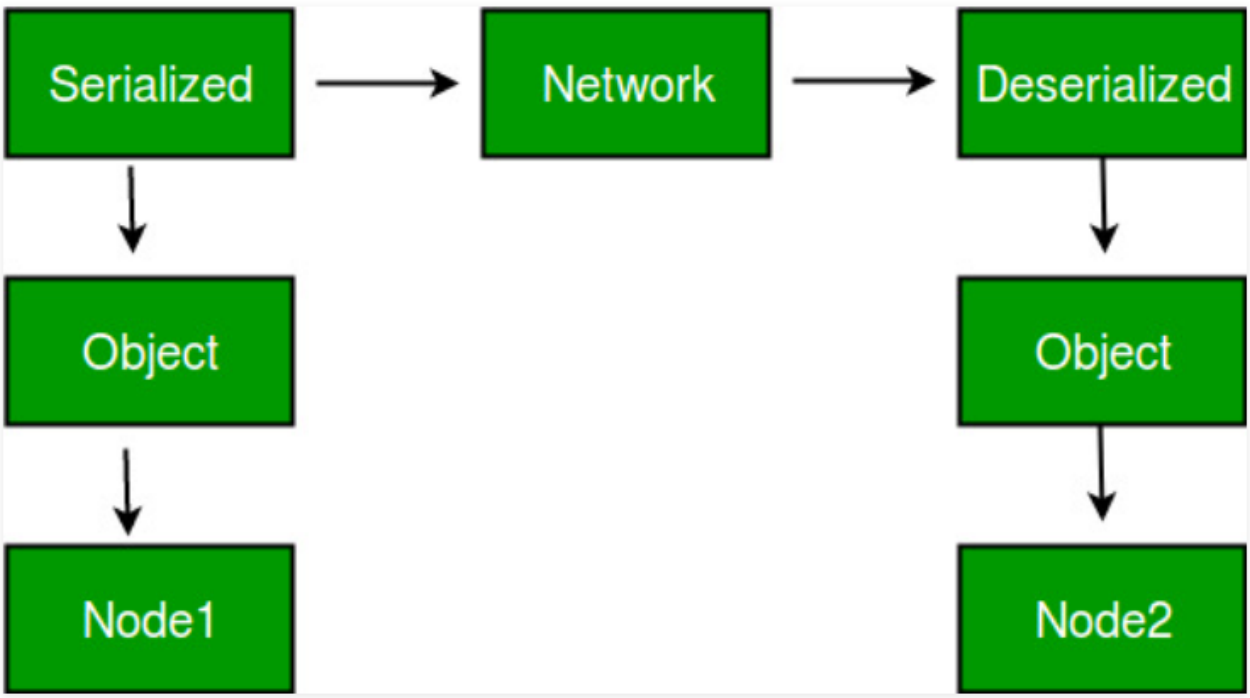
17. Serialize & Deserialize

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream i![Sử dụng CyclicBarrier trong Java - GP Coder (Lập trình Java)](https://gpcoder.com/wp-content/uploads/2018/03/CyclicBarrier.png)s used to recreate the actual Java object in memory. This mechanism is used to persist the object.



Advantages of Serialization

1. To save/persist state of an object.
2. To travel an object across a network.



Points to remember

- If a parent class has implemented **Serializable** interface then child class doesn't need to implement it but vice-versa is not true.
- Only **non-static data members** are saved via **Serialization process.**
- **Static data members** and **transient data members** are not saved via Serialization process.So, if you don't want to save value of a non-static data member then make it transient.
- **Constructor of object is never called** when an object is deserialized.
- **Associated objects** must be implementing Serializable interface.

SerialVersionUID

`The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the receiver has loaded a class for the object that has different UID than that of corresponding sender's class, the Deserialization will result in an **InvalidClassException**. A Serializable class can declare its own UID explicitly by declaring a field name. It must be static, final and of type long.`

```
public static void main(String[] args) {
    Dummy dummy = new Dummy();
    String file = "dummy.ser";
    // Serialize
    try(FileOutputStream fileOutput = new FileOutputStream(file);
        ObjectOutputStream objectOut = new ObjectOutputStream(fileOutput);) {
        // Saving of object in a file
        objectOut.writeObject(dummy);
        System.out.println("Object has been serialized\n"
                           + "Data before Deserialization.");
        System.out.println(dummy.toString());
    } catch (IOException fileNotFoundException) {
        fileNotFoundException.printStackTrace();
    }
    // Deserialize
    try(FileInputStream fileInput = new FileInputStream(file);
        ObjectInputStream objectInput = new ObjectInputStream(fileInput)){
        Dummy dummy1 = (Dummy) objectInput.readObject();
        System.out.println("Object has been deserialized\n"
                + "Data after Deserialization.");
        System.out.println(dummy1.toString());
    } catch (IOException | ClassNotFoundException fileNotFoundException) {
        fileNotFoundException.printStackTrace();
    }
}
```