FOUNDAMENTAL PROGRAMMING STRUCTURES

Java is a **compiled language** (you write the source code and you compile it).

The **compiler** is a program that translates the source code into **object code**, that is, code that can be executed by a machine.

Java is **independent from the machine** (interoperability), because the compiler translates the source code in an object code understandable not by the machine directly but by a **Virtual Machine** (**Java virtual machine**).

1.1. A FIRST PROGRAM

```
public class prova {
   public static void main (String[] args {
           System.out.println("3+2 equals:");
           System.out.println(3+2);
   }
}
```

The output will be:
```
3+2 equals:
5
```

This program is very simple and prints whatever is in the braces of the `println` method. Mind that `println` will print and go to new line so if we want to keep it in the same line we do:

```
public class prova {
   public static void main (String[] args {
           System.out.print("3+2 equals: ");
           System.out.println(3+2);
   }
}
```

The output will be:
```
3+2 equals: 5
```

How can we make this so that it works for any two integers x and y?

```
public class prova {
   public static void main (String[] args {
           System.out.print("3+2 equals: ");
           doIt(3, 2);
   }
   public static void doIt (int x, int y) {
           System.out.println(x + y);
   }
}
```

The output will be:
```
3+2 equals: 5
```

When the program is run, the compiler knows the first piece of code to be executed is the `main` w hich must be declared as `static`.

Remember that when we declare a variable inside a scope, that variable only exist within that scope.

1.2. PRIMITIVE TYPES AND WRAPPER CLASSES

Primitive types are the most basic data types in Java. They represent single values and are not objects (so they don't have methods). There are eight primitive types in Java:

- **byte:** 8-bit signed integer.
- **short**: 16-bit signed integer.
- **int**: 32-bit signed integer.
- **long**: 64-bit signed integer.
- **float**: Single-precision 32-bit floating-point number.
- **double**: Double-precision 64-bit floating-point number.
- **char**: 16-bit Unicode character.
- **boolean**: Represents a boolean value (**true** or **false**).

Wrapper classes in Java are classes that encapsulate primitive types, allowing them to be treated as objects. Each primitive type has a corresponding wrapper class:

- **Byte**: Wrapper for *byte*.
- **Short**: Wrapper for *short*.
- **Integer**: Wrapper for *int*.
- **Long**: Wrapper for *long*.
- **Float**: Wrapper for *float*.
- **Double**: Wrapper for *double*.
- **Character**: Wrapper for *char*.
- **Boolean**: Wrapper for *boolean*.

Wrapper classes provide utility methods for converting, parsing, and performing operations on primitive types. They also allow primitive types to be used in contexts where objects are required.

In fact, in Java, generic classes allow you to create classes and methods that can operate on objects of any data type. However, primitive types cannot be directly used as type parameters in generic classes. For example, you cannot create an **ArrayList<int>** because **int** is a primitive type.
To work around this limitation, you can use the corresponding wrapper classes for primitive types. For example, you can use **ArrayList<Integer>** instead of **ArrayList<int>**.

1.2.1. AUTOBOXING AND UNBOXING

Java provides automatic conversion between primitive types and their corresponding wrapper classes, known as **autoboxing** and **unboxing**.

- **Autoboxing:** The process of converting a primitive type to its corresponding wrapper class object automatically.
- **Unboxing:** The process of extracting the primitive value from the wrapper class object automatically.

Example:

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(42);
int first = numbers.get(0);
```

In the example above, when you add an **int** value (2nd line) to an **ArrayList<Integer>**, Java automatically converts it to an **Integer** object (*autoboxing*). Similarly, when you retrieve an **Integer** object from the **ArrayList** and assign it to an **int** variable (3rd line), Java automatically extracts the **int** value from the **Integer** object (*unboxing*).

1.2.2. COMPARISON

When working with wrapper objects, you need to be cautious when using the **==** and **!=** operators for comparison. These operators compare object references, not the contents of objects. To compare the contents of wrapper objects, you should use the **equals** method.

```
if (numbers.get(i).equals(numbers.get(j))) {
    //body of the if
}
```

1.3. INITIALIZATION AND DECLARATION FOR VARIABLES

Variable **declaration** is the process of specifying the data type and name of a variable. In Java, variables must be declared before they can be used. The general syntax for variable declaration is:

```
dataType variableName;
```

Variable **initialization** is the process of assigning an initial value to a variable at the time of declaration:

```
dataType variableName = initialValue;
```

or later in the program:

```
variableName = newValue;
```

If a variable is not initialized, it will have a default value based on its data type.

1.3.1. CONSTANTS

The **final** keyword denotes a value that cannot be changed once it has been assigned. In other languages, one would call such a value a constant.

For example:

```
final int DAYS_PER_WEEK = 7;
```

By convention, uppercase letters are used for names of constants.

You can also declare a constant outside a method, using the **static** keyword:

```
public class Calendar {
    public static final int DAYS_PER_WEEK = 7;
    ...
}
```

Attenzione

final indica che il valore della variabile non può essere modificato dopo l'inizializzazione.

Quando viene utilizzato per dichiarare una variabile, **static** indica che la variabile è condivisa tra tutte le istanze della classe. La variabile statica esiste una sola volta nella memoria, indipendentemente dal numero di istanze della classe create.

1.3.2. VARIABLE SCOPE

The **scope** of a variable is the part of the program where you can access the variable.

When we declare a variable we can either do it inside of a method, that is, as a **local variable** or at the top level of a class, that is, as an **instance variable**.

- **Instance variables** are accessible to all methods within the class and can be accessed by any instance (object) of the class. Their lifetimes are tied to the lifetime of the object (instance) of the class.
- **Local variables** are accessible only within the method in which they are declared and have a limited scope that extends from their declaration to the end of the block in which they are declared (in practice the curly brackets). Their lifetimes are limited to the execution of the method in which it is declared.

1.4. CONDITIONAL STATEMENTS SYNTAX

The conditional statements in java are similar to the ones In python:

```
if (condition1) {
    // Code to execute if condition1 is true ;
} else if (condition2) {
    // Code to execute if condition2 is true ;
} else {
    // Code to execute if neither condition1
       nor condition2 is true                ;
}
```

Remember that the semicolon (;) is not just required at the end of each line, it is a statement itself. Anytime we want a piece of code (say the body of an if) not to do anything, we put the semicolon.

1.5. ITERATIVE STATEMENTS SYNTAX

The conditional statements in java are similar to the ones In python:

FOR LOOP: used to iterate a block of code a **fixed number of times**. It consists of three parts: initialization, condition, and update.

```
for (initialization; condition; update) {
    // Code to execute repeatedly
}
```

WHILE LOOP: used to iterate a block of code **as long as a specified condition is true**.

```
while (condition) {
    // Code to execute repeatedly
}
```

DO WHILE LOOP: similar to the while loop, but the condition is checked after executing the block of code, so it always executes the block at least once.

```
do {
    // Code to execute repeatedly
} while (condition);
```

FOREACH: (also known as the enhanced for loop) is used to iterate over elements of arrays or collections.

```
for (type var : array/collection) {
    // Code to execute for each element
}
```

Remember

Java has increment and decrement operators:

```
n++; // Adds one to n
n—; // Subtracts one from n
```

So you can use these to handle the indexes to iterate on

1.5.1. BREAKING AND CONTINUING

If you want to exit a loop in the middle, you can use the **break** statement. When the break statement is reached, the loop is exited immediately.

If you want to jump to the end of another enclosing statement, use a **labeled break statement**:

```
outer:
while (...) {
       ...
       while (...) {
              ...
              if (...) break outer;
              ...
       }
       ...
}
// Labeled break jumps here
```

You label the top of the statement, but the break statement jumps to the end.

The **continue** statement is similar to break, but instead of jumping to the end of the loop, it jumps to the end of the current loop iteration.
In a for loop, the continue statement jumps to the next update statement.

1.6. ARRAYS AND ARRAY LISTS

Arrays are a fundamental programming construct for collecting multiple items of the same type. Java has array types built into the language, and it also supplies an **ArrayList** class for arrays that grow and shrink on demand.

1.6.1. WORKING WITH ARRAYS

For every type, there is a corresponding array type. An array of integers has type **int[]**, an array of String objects has type **String[]**, and so on.

Example:

```
String[] names; // creates an array of strings
                   called names
```

The variable isn't yet initialized. Let's initialize it with a new array. For that, we need the new operator:

```
names = new String[100];
```

Of course, we can combine these two statements:

```
String[] names = new String[100];
```

Now names refers to an array with 100 elements, which you can access as names[0] ... names[99].

When you construct an array with the **new** operator, it is filled with a **default value**.

- Arrays of numeric type (including char) are filled with zeroes.
- Arrays of boolean are filled with false.
- Arrays of objects are filled with null references.

You can then fill the array with values by writing a loop. However, sometimes you already know the values that you want, and you can just list them inside braces:

```
int[] primes = { 2, 3, 5, 7, 11, 13 };
```

In this case, you don't use the new operator, and you don't specify the array length.

1.6.2. ARRAY LISTS

When you construct an array, you need to know its length. Once constructed, the length can never change. That is inconvenient in many practical applications.

A remedy is to use the ArrayList class in the **java.util package**. An ArrayList object manages an array internally. When that array becomes too small or is insufficiently utilized, another internal array is automatically created, and the elements are moved into it. This process is invisible to the programmer using the array list.

Array lists are classes, and you use the normal syntax for constructing instances and invoking methods. However, unlike the classes that you have seen so far, the ArrayList class is a **generic class** — a class with a type parameter.

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

There are no construction arguments in this call, but it is still necessary to supply the () at the end.

The result is an array list of size 0. You can add to the end and remove elements with the add and remove method:

```
friends.add(“Peter”);
friends.add(“Paul”);
friends.add(0, “Paul”); // Adds before index 0
friends.remove(1);
```

To access elements use method calls instead of [ ] syntax. **get** method reads an element, **set** method replaces an element with another and **size** method yields the current size of the list.

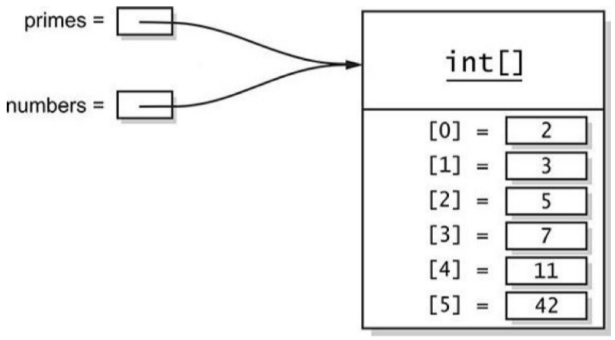
Example:

```
for (int i = 0; i < friends.size(); i++) {
      System.out.println(friends.get(i));
}
```

1.6.3. COPYING ARRAYS AND ARRAY LISTS

You can copy one array variable into another, but then both variables will refer to the same array.

```
int [] numbers = primes;
numbers[5]  = 42; //now primes[5] is also 42
```



If you don’t want this sharing, you need to make a copy of the array. To do so, you can use the static **Arrays.copyOf** method.

```
int[] copiedPrimes = Arrays.copyOf(primes,
                     primes.lenght)
```

This method constructs a new array of the desired length and copies the elements of the original array into it.

The same thing holds for array lists, to copy an array list:

```
ArrayList<String> copiedFriends = new ArrayList
<>(friends);
```

1.6.4. MULTIDIMENSIONAL ARRAYS

Java does not have true multidimensional arrays. They are implemented as arrays of arrays.

OBJECT-ORIENTED PROGRAMMING

In object-oriented programming, work is carried out by collaborating objects whose behaviour is defined by the classes to which they belong.

In Java, every method is declared in a class and, except for a few primitive types, every value is an object.

Encapsulation, a foundational concept in object-oriented programming, ensures that users can interact with objects without needing to understand their internal workings. When utilizing objects developed by others, invoking their methods shields users from needing insight into the underlying complexities.

2.1. ACCESSOR AND MUTATOR METHODS

A method is said **mutator** if it changes the object on which it was invoked.

By contrast, it is said **accessor** if it leaves the object unchanged.

2.2. OBJECT REFERENCES

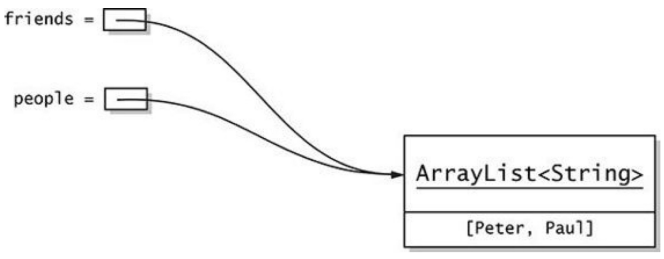
In some programming languages (such as C++), a variable can actually hold the object (that is, the actual bits that make up the object's state).

In Java, that is not the case. A variable can only hold a **reference** to an object. The actual object is elsewhere and the reference is some implementation-dependent way of locating the object.

When you assign a variable holding an object reference to another, you have two references to the SAME object.

```
ArrayList<String> people = friends;
//Now people and friends refer to the same
  object
```

If you mutate the shared object, the **mutation is visible through both references**.



Most of the time, this sharing of objects is efficient and convenient, but you have to be aware that it is **possible to mutate a shared object through any of its references**. However, if a class has no mutator methods, you can freely give out references to one of its objects as nobody can change it.

Finally, it possible for an object variable to refer to no object at all, by setting it to the value **null**.

When an object is no longer needed, eventually the **garbage collector** will recycle the memory and make it available for reuse. In Java, this process is completely automatic.

2.3. IMPLEMENTING CLASSES

Let's use an example to see how to implement a class. We will model an employee class.
An employee has a *name* and a *salary*. In this example the name can't change but the salary can be increased.

2.3.1. INSTANCE VARIABLES

From the description of employee objects, you can see that the state of such an object is described by two values: *name* and *salary*.

In Java, **instance variables** are used to describe the state of an object, every object or *instance* of the class has those variables. They are declared in a class like this:

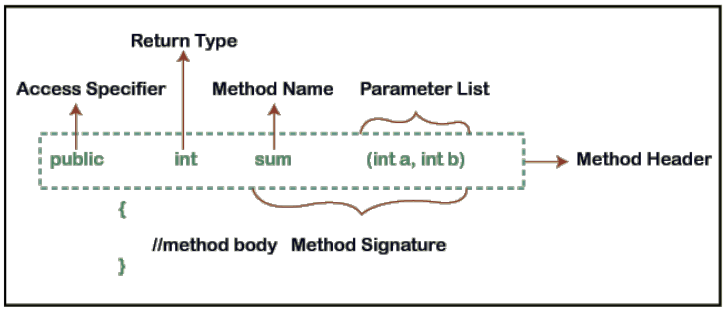
```
public class Employee {
   private String name;
   private double salary;
   …
}
```

That means that every object or *instance* of the Employee class has these two variables.

Instance variables are usually declared as **private**. That means that only methods of the same class can access them. When a variable is private, you control which parts of your program can modify the variables and you can decide at any point to change the internal representation.

2.3.2. METHOD DECLARATION

When we declare a method, the syntax is:



- **Return Type:** The method must return the type specified. If we don't want our method to return anything, the return type is **void**.
- **Method Name:** The method name should be descriptive of the function the method performs.
- **Method Parameters:** They are declared within parentheses. Each parameter consists of a type and a name separated by a space.
- **Method Body:** The method body contains the statements that the method will execute when called. It's enclosed within curly braces **{ }**.
- **Method Call:** To call a method, use the method name followed by parentheses. If the method requires

parameters, they are passed during the call within the parentheses.

Example:

```
public void raiseSalary(double byPercent)
```

This method receives a parameter of type double and doesn't return any value, as indicated by the return type void.

```
public String getName()
```

This method has no parameters and returns a string.

Most methods are declared as **public**, which means anyone can call such a method. Sometimes, a helper method is declared **private**, which restricts it to being used only in other methods of the same class.

2.3.3. METHOD BODIES

Following the method header, you provide the body:

```
public void raiseSalary(double byPercent) {
   double raise = salary * byPercent / 100;
   salary += raise;
}
```

Use the return keyword if the method yields a value:

```
public String getName() {
   return name;
}
```

Obviously, the methods are declared inside the class declaration.

2.3.4. INSTANCE METHOD INVOCATIONS

Consider this example of a method call:

```
fred.raiseSalary(5);
```

In this call, the argument *5* is used to initialize the parameter variable *byPercent*, the method then operates on an instance of the class *Employee*.

Therefore, such a method is called an **instance method**. In Java, all methods that are not declared as static are instance methods.

As you can see, two values are passed to the raiseSalary method: a reference to the object on which the method is invoked, and the argument of the call. Technically, both of these are parameters of the method, but in Java, as in other object oriented languages, the first one takes on a special role. It is sometimes called the **receiver of the method call**.

2.3.5. THE this REFERENCE

When a method is called on an object, **this** is set to that object. If you like, you can use the *this* reference in the implementation.

```
public void raiseSalary(double byPercent) {
   double raise = this.salary * byPercent/100;
   this.salary += raise;
}
```

It is now obvious that *raise* is a local variable and *salary* is an instance variable.

2.3.6. CALL BY VALUE

When you pass an object to a method, the method obtains a **copy of the object reference**. Through this reference, it can access or mutate the parameter object.

```
public class EvilManager {
   private Random generator;
   ...
   public void giveRandomRaise(Employee e {
      double percentage = 10 *
      generator.nextGaussian();
      e.raiseSalary(percentage);
   }
}
```

In this case, because the object passed as parameter is an Employee, when we call:

```
boss.giveRandomRaise(fred);
```

The salary of Fred is actually changed.

In Java, **primitive types** are passed by value to methods. This means that when a primitive type parameter is passed to a method, a copy of its value is made and provided to the method. Any modifications made to the parameter within the method only affect the local copy of the parameter and do not have any impact on the original value outside the method.

```
public void increaseRandomly(double x) {
   double amount = x * generator.nextDouble();
   x += amount;
}
```

In this case, because the object passed as parameter is a double, when we call:

```
boss.increaseRandomly(sales);
```

x is increased but then goes out of scope and sales remains unchanged.

For the same reason, when **object references** (i.e., references to objects created using the new keyword) are passed to a method, a copy of the reference is made and provided to the method. This copy of the reference still points to the same object in memory. Therefore, if you attempt to reassign the reference within the method to point to a different object, it only affects the local copy of the reference and does not change the original reference outside the method (when the copy variable goes out of scope).

```
public void replaceWithZombie(Employee e) {
   e = new Employee("", 0);
}
```

When we call:

```
boss.replaceWithZombie(fred);
```

At no point Fred was changed so when the method exits, no changes are performed.

2.4. MAIN METHOD

The **main** method in Java serves as the entry point for the execution of a Java program. It has a fixed structure that adheres to Java's syntax rules.

```
public static void main(String[] args) {
```

```
    // Method body
}
```

- **public**: can be accessed from outside the class.
- **static**: can be called without creating an instance of the class.
- **void**: does not return any value.
- **main**: The name of the method is main.
- **String[] args**: accepts a single parameter of type *String[]*, which represents an array of strings. This parameter allows the passing of command-line arguments to the Java program.

2.5. OBJECT CONSTRUCTION

One step remains to complete the Employee class: We need to provide a **constructor**. Every class has **at least one constructor.**

2.5.1. IMPLEMENTING CONSTRUCTORS

Declaring a **constructor** is similar to declaring a method. However, the name of the constructor is the same as the class name, and there is no return type.

```
public Employee(String name, double salary {
   this.name = name;
   this.salary = salary;
}
```

Note This constructor is public. It can also be useful to have private constructors.

A constructor executes when you use the **new operator**. Example:

```
new Employee("James Bond", 500000)
```

allocates an object of the Employee class and invokes the constructor body, which sets the instance variables to the arguments supplied in the constructor.

The *new* operator returns a reference to the constructed object. You will normally want to save that reference in a variable:

```
Employee james = new Employee("James Bond",
                              500000);
```

or pass it to a method:

```
ArrayList staff = new ArrayList<>();
staff.add(new Employee("James Bond", 500000));
```

2.5.2. OVERLOADING

You can supply more than one version of a constructor. For example, if you want to make it easy to model nameless workers, supply a second constructor that only accepts a salary.

```
public Employee(double salary) {
   this.name = ""; this.salary = salary;
}
```

Now the Employee class has two constructors. Which one is called depends on the arguments.

In this case, we say that the constructor is **overloaded**.

2.5.3. DEFAULT INITIALIZATION

If you don't set an instance variable explicitly in a constructor, it is automatically set to a **default value**: numbers to 0, boolean values to false, and object references to null.

```
Employee(String name) {
   this.name = name;
}
```

In this regard, instance variables are very different from local variables. Recall that you must always explicitly initialize local variables.

2.5.4. INSTANCE VARIABLE INITIALIZATION

You can specify an initial value for any instance variables, like this:

```
public class Employee {
   private String name = "";
   ...
}
```

This initialization occurs after the object has been allocated and before a constructor runs. Therefore, the initial value is present in all constructors. Of course, some of them may choose to overwrite it.

2.5.5. FINAL INSTANCE VARIABLE

You can declare an instance variable as **final**.

Such a variable must be initialized by the end of every constructor. Afterwards, the variable may not be modified again.

For example, the name variable of the Employee class may be declared as final because it never changes after the object is constructed (there is no setName method).

```
public class Employee {
   private final String name;
   ...
}
```

When used with a reference to a mutable object, the final modifier merely states that the reference will never change. It is perfectly legal to mutate the object.

```
public class Person {
   private final ArrayList friends = new
   ArrayList<>();
   ...
}
```

Methods may mutate the array list to which friends refers, but they can never replace it with another. In particular, it can never become null.

2.6. STATIC VARIABLES AND METHODS

2.6.1. STATIC VARIABLES

If you declare a variable in a class as **static**, then there is only one such variable per class. (In contrast, each object has its own copy of an instance variable).

For example, suppose we want to give each employee a distinct ID number. Then we can share the last ID that was given out:

```
public class Employee {
   private static int lastId = 0;
   private int id;
   ...
   public Employee() {
      lastId++;
      id = lastId;
   }
}
```

Every Employee object has its own instance variable id, but there is only one lastId variable that belongs to the class, not to any particular instance of the class.

When a new Employee object is constructed, the shared lastId variable is incremented and the id instance variable is set to that value. Thus, every employee gets a distinct id value.

2.6.2. STATIC CONSTANTS

Static constants (that is, **static final** variables) are quite common.

For example, the Math class declares a static constant:

```
public class Math {
  ...
  public static final double PI = 3.1415926535;
  ...
}
```

You can access this constant in your programs as Math.PI.

Without the static keyword, PI would have been an instance variable of the Math class. That is, **you would need an object of the class to access PI**, and every Math object would have its own copy of PI.

2.6.3. STATIC METHODS

Static methods are methods that **do not operate on objects**.

For example, the pow method of the Math class is a static method. The expression Math.pow(x, a) computes the power x^a. It does not use any Math object to carry out its task.

```
public class Math {
  public static double pow(double base, double
                                    exponent){
     ...
  }
}
```

- **Why not make pow into an instance method?** It can't be an instance method of double since, in Java, primitive types are not classes.
- **Why not make it an instance method of the Math class?** Then you would need to construct a Math object in order to call it.

Since static methods don't operate on objects, you cannot access instance variables from a static method. However, static methods can access the static variables in their class.

2.7. OBJECT CONSTRUCTION

In Java, you place related classes into a **package**. Packages are convenient for organizing your work and for separating it from code libraries provided by others.

The main reason for using packages is to guarantee the *uniqueness of class names*. As long as all of them place their classes into different packages, there is no conflict.

2.7.1. PACKAGE DECLARATION

A **package name** is a dot-separated list of identifiers (such as *java.util.regex*).

In Java, packages **do not nest**. For example, the packages *java.util* and *java.util.regex* have nothing to do with each other. Each is its own independent collection of classes.

To place a class in a package, you add a **package statement** as the first statement of the source file:

```
package com.horstmann.corejava;
public class Employee {
   ...
}
```

Now, the Employee class is in the *com.horstmann.corejava* package, and its fully qualified name is *com.horstmann.corejava.Employee*.

2.7.2. PACKAGE SCOPE

You have already encountered the access modifiers **public** and **private**. Features tagged as public can be used by any class. Private features can be used only by the class that declares them.

If you don't specify either public or private, the feature (that is, the class, method, or variable) can be accessed by all methods in the **same package**.

Package scope is useful for utility classes and methods that are needed by the methods of a package but are not of interest to the users of the package. Another common use case is for testing. You can place test classes in the same package, and then they can access internals of the classes being tested.

2.7.3. IMPORTING CLASSES

The **import statement** lets you use classes without the fully qualified name. For example, when you use *import java.util.Random*; then you can write *Random* instead of *java.util.Random* in your code.

You can import all classes from a package with a **wildcard**:

```
import java.util.*;
```

The wild card can only import classes, not packages. You cannot use import *java.**; to obtain all packages whose name starts with *java*.

2.7.4. STATIC IMPORTS

A form of the *import* statement permits the importing of static methods and variables.

For example, if you add the **directive**:

```
import static java.lang.Math.*;
```

to the top of your source file, you can use the static methods and static variables of the Math class without the class name prefix:

```
sqrt(pow(x, 2) + pow(y, 2))
// i.e., Math.sqrt, Math.pow
```

You can also import a specific static method or variable:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.PI;
```

2.8. NESTED CLASSES

In the preceding section, you have seen how to organize classes into packages. Alternatively, you can place a class inside another class. Such a class is called a **nested class**.

Java has two kinds of nested classes, with somewhat different behavior.

2.8.1. STATIC NESTED CLASSES

Consider an *Invoice* class that bills for items, each of which has a description, quantity, and unit price. We can make *Item* into a nested class:

```
public class Invoice {
   private static class Item {
      // Item is nested inside Invoice
      String description;
      int quantity;
      double unitPrice;
      double price() {
            return quantity * unitPrice;
      }
   }
   private ArrayList items = new ArrayList<>();
   ...
}
```

There is nothing special about the Item class, except for **access control**.

The class is **private** in *Invoice*, so only Invoice methods can access it. For that reason, we did not bother making the instance variables of the inner class private.

Here is an example of a method that constructs an object of the nested class:

```
public class Invoice {
   ...
   public void addItem(String description, int
   quantity, double unitPrice) {
      Item newItem = new Item();
      newItem.description = description;
      newItem.quantity = quantity;
      newItem.unitPrice = unitPrice;
      items.add(newItem);
   }
}
```

2.8.2. INNER CLASSES

A nested class is declared as *static*.
If you drop the static modifier, such classes are called **inner classes**.

Consider a social network in which each member has friends that are also members.

```
public class Network {
   public class Member {
   // Member is an inner class of Network
      private String name;
      private ArrayList friends;

      public Member(String name) {
         this.name = name;
         friends = new ArrayList<>();
      }
      ...
   }
   private ArrayList members;
   ...
}
```

With the static modifier dropped, there is an essential difference. A *Member* object knows to which network it belongs.
Let's see how this works. First, here is a method to add a member to the network:

```
public class Network {
   ...
   public Member enroll (String name) {
      Member newMember = new Member(name);
      members.add(newMember);
      return newMember;
   }
}
```

If we call:

```
Network myFace = new Network();
Network.Member fred = myFace.enroll("fred");
```

We create a new network called myFace and we added a member Fred. Now let's assume that Fred thinks this isn't the hottest social network anymore, and he wants to leave. Here is the implementation of the leave method:

```
public class Network {
   public class Member {
      ...
      public void leave() {
         members.remove(this);
      }
   }
   private ArrayList<Member> members;
   ...
}
```

As you can see, **a method of an inner class can access instance variables of its outer class**. In this case, they

are the instance variables of the outer class object that created it, the unpopular myFace network.

This is what makes an inner class different from a static nested class. **Each inner class object has a reference to an object of the enclosing class.**

- Use a **static nested class** when the instances of the nested class don't need to know to which instance of the enclosing class they belong.
- Use an **inner class** only if this information is important. An inner class can also invoke methods of the outer class through its outer class instance.

2.8.3. SPECIAL SYNTAX RULES FOR INNER CLASSES

The expression *OuterClass.this* denotes the **outer class reference**. For example, you can write the leave method of the Member inner class as:

```
public void leave() {
   Network.this.members.remove(this);
}
```

INTERFACES AND LAMBDA EXPRESSIONS

3.1. INTERFACES

Interfaces serve as contracts between service providers and classes utilizing the service.
They define the methods that classes must implement to ensure compatibility with the service.

In Java, interfaces are a fundamental part of the language, facilitating the creation of flexible and modular code.

3.1.1. DECLARING AN INTERFACE

When declaring an interface, the focus is on **specifying the method headers without providing implementation details**. This allows for various classes to implement the interface according to their specific requirements.

For example, consider a scenario where we need to work with different types of sequences of integers. These sequences could be provided by users, generated randomly, or derived from other sources such as arrays or strings.
To address this variability, we define an interface named *IntSequence*.

```
public interface IntSequence {
    boolean hasNext();
    int next();
}
```

This interface includes two essential methods: *hasNext()* to test whether there is a next element in the sequence, and *next()* to retrieve the next element. By declaring these methods in the interface, we establish a c**ommon framework** that any class implementing the interface must adhere to.

In Java, interfaces can include **default implementations** for methods, although it's not mandatory. When no implementation is provided, the method is considered **abstract**, indicating that implementing classes must define their own implementations.

The key advantages of interfaces lie in their flexibility and interoperability. Once an interface is defined, any class that implements it can be used interchangeably with methods or services expecting objects of that interface type. For instance, we can utilize the *IntSequence* interface to implement a method *average* that calculates the average of the first n values in a sequence. This method operates seamlessly with any class that implements the *IntSequence* interface, regardless of the specific implementation details of those classes.

3.1.2. IMPLEMENTING AN INTERFACE

Implementing an interface in Java involves creating a class that provides **concrete implementations** for all the methods declared in the interface.

Consider a class named *SquareSequence* that implements the *IntSequence interface*. This class represents a sequence of square numbers, where each subsequent number is the square of the next integer. Here's how the SquareSequence class is implemented:

```
public class SquareSequence implements
IntSequence {
   private int i;

   public boolean hasNext() {
      return true;
    }

   public int next() {
      i++;
      return i * i;
    }
}
```

The SquareSequence class provides implementations for both methods specified in the IntSequence interface:

- The hasNext() method always returns true, indicating that there is always a next element in the sequence.
- The next() method increments the value of i, calculates the square of i, and returns the result.

The **implements** keyword in the class declaration indicates that the SquareSequence class intends to adhere to the contract specified by the IntSequence interface.

It's important to note some considerations when implementing interfaces in Java:

- All methods from the interface must be declared as **public** in the implementing class.
- If a class implements an interface but does not provide implementations for all methods declared in the interface, the class must be declared as **abstract**.

3.1.3. CONVERTING TO AN INTERFACE TYPE

Consider the following code fragment, which computes the average of a sequence of digit values:

```
IntSequence digits = new DigitSequence(1729);
double avg = average(digits, 100);
```

In this fragment, the variable digits is of type *IntSequence*, not *DigitSequence*.

The IntSequence type refers to an object of a class that implements the IntSequence interface.

The DigitSequence class is a concrete implementation of the IntSequence interface.

By assigning a DigitSequence object to a variable of type IntSequence, we can treat the DigitSequence object as an IntSequence, facilitating flexibility in code design and usage.

In this context, it's useful to understand the terminology of **supertype** and **subtype**:

- A **supertype** (S) is a more general type that encompasses one or more subtypes.
- A **subtype** (T) is a specialized type that inherits or implements the characteristics of its supertype.

In this example, IntSequence is the supertype, and DigitSequence is the subtype.

Important Note

While it's possible to declare variables of an interface type, it's essential to remember that **objects themselves cannot have interface types**. All objects are instances of concrete classes.

3.1.4. CASTS AND THE instanceof OPERATOR

Casts — Converting from Supertype to Subtype

Occasionally, there arises a need to convert **from a supertype to a subtype**. This typically happens when you want to access specific methods or properties defined in the subtype. Here's an example:

```
IntSequence sequence = ...; // Some IntSequence
                               object
DigitSequence digits = (DigitSequence)sequence;
System.out.println(digits.rest());
```

In this code snippet, sequence is of type IntSequence, but we know that it's actually a DigitSequence. Hence, we perform a cast to convert it to the subtype DigitSequence. This allows us to access the rest() method, which is specific to the DigitSequence class.

Important Considerations: You can only cast an object to its actual class or one of its supertypes. If the cast is incorrect, it may result in a compile-time error or a class cast exception at runtime.

Using the instanceof Operator

To avoid runtime errors when casting objects, it's prudent to first check whether the object is of the desired type using the **instanceof operator**. This operator evaluates to true if the object is an instance of a class that has the specified type as a supertype. Here's the syntax:

```
if (object instanceof Type) {
    // Perform the cast safely
    Type typedObject = (Type) object;
    // Proceed with using typedObject
}
```

3.1.5. EXTENDING INTERFACES

An interface can **extend another interface**, thereby inheriting its methods while adding additional ones.

For instance, consider the Closeable interface, which contains a single method close() used for resource cleanup.

```
public interface Closeable {
    void close();
}
```

Building upon *Closeable*, the *Channel* interface extends it and introduces an additional method *isOpen()*:

```
public interface Channel extends Closeable {
    boolean isOpen();
}
```

Any class implementing Channel must provide implementations for both methods, thus ensuring adherence to both interfaces.

3.1.6. IMPLEMENTING MULTIPLE INTERFACES

In Java, a class can implement multiple interfaces, enabling it to **exhibit behaviors defined by each interface**. For instance, consider a class named FileSequence responsible for reading integers from a file. This class can implement both IntSequence and Closeable interfaces:

```
public class FileSequence implements
IntSequence, Closeable {
    // Implementations for IntSequence and
       Closeable methods
}
```

By implementing both IntSequence and Closeable, the FileSequence class can function as a sequence of integers while also providing resource management capabilities for closing the file.

3.1.7. CONSTANTS

In Java, any variable defined within an interface is automatically considered as **public static final**, making them constants. These constants can be accessed using the interface's qualified name or directly within classes implementing the interface.

For instance, let's examine the SwingConstants interface, which provides constants for compass directions:

```
public interface SwingConstants {
    int NORTH = 1;
    int NORTH_EAST = 2;
    int EAST = 3;
    // Other constants...
}
```

In this interface, NORTH, NORTH_EAST, EAST, and any other variables defined are treated as public static final constants. Thus, they can be accessed using the interface's qualified name, such as *SwingConstants.NORTH*.

If a class chooses to implement the SwingConstants interface, it can directly use the constants without qualifying them with the interface name.

It's important to note that **interfaces cannot contain instance variables**. They specify behavior rather than object state. Therefore, constants defined within interfaces are static and immutable, serving as values that remain constant across all instances of classes implementing the interface.

3.2. STATIC AND DEFAULT METHODS

In earlier days of Java, all methods of an interface had to be abstract — that is, without a body. Nowadays you can add two kinds of methods with a concrete implementation: **static** and **default** methods.

3.2.1. STATIC METHODS

Static methods in interfaces can be invoked using the interface's name, making them accessible without needing an instance of a class that implements the interface.

For example, consider the *IntSequence* interface with a static method *digitsOf*:

```
public interface IntSequence {
    ...
    public static IntSequence digitsOf(int n) {
        return new DigitSequence(n);
    }
}
```

Here, the digitsOf method generates a sequence of digits for a given integer and returns an instance of some class implementing the IntSequence interface. This approach enables the interface to encapsulate the creation logic of sequences without exposing the concrete class to the caller.

3.2.2. DEFAULT METHODS

Default methods in interfaces provide a default implementation for methods. These methods are marked with the **default** modifier and can be overridden by classes implementing the interface.

For instance, consider the *IntSequence* interface with a default implementation for the *hasNext* method:

```
public interface IntSequence {
    default boolean hasNext() {
        return true;
        // By default, sequences are infinite
    }
    int next();
}
```

n this example, classes implementing the IntSequence interface can choose to override the *hasNext* method or **inherit the default implementation**.

Without default methods, adding non-default methods to an interface would not be source-compatible, potentially causing compilation errors in classes implementing the interface.

3.2.3. RESOLVING DEFAULT METHODS CONFLICTS

In Java, when a class implements multiple interfaces, and there is a **conflict between default methods with the same name and parameter types** in those interfaces, the conflict **needs to be resolved explicitly**. This scenario is rare but can occur, and resolving the conflict is straightforward.

Suppose we have two interfaces, *Person* and *Identified*, both of which define a default method *getId()*:

```
public interface Person {
    String getName();
    default int getId() { return 0; }
}
```

```
public interface Identified {
    default int getId() { return hashCode(); }
}
```

Now, if a class Employee implements both Person and Identified, it inherits two conflicting *getId* methods. To resolve this conflict, the class must provide its own implementation of the getId method:

```
public class Employee implements Person,
Identified {
    public int getId() {
        return Identified.super.getId();
        // Delegate to Identified interface's
           getId method
    }
    // Other implementations...
}
```

In this example, the Employee class explicitly overrides the getId method and chooses to delegate the implementation to the getId method of the Identified interface using the **super keyword**. This resolves the ambiguity and ensures that the correct method is invoked.
Using the super keyword allows you to **call a method from a supertype explicitly**.

If the Identified interface does not provide a default implementation for the getId() method, and the Employee class implements both Person and Identified interfaces, the Employee class can inherit the default method from the Person interface. In this scenario, there is no conflict between default methods, as the Identified interface does not define a default implementation for getId().

However, it's essential to note that even though the Employee class inherits the default method from the Person interface, **it must still provide its own implementation for the getId() method if it intends to be concrete**. Otherwise, it should be declared as **abstract**.

3.3. EXAMPLE OF INTERFACES

3.3.1. COMPARABLE INTERFACE

When sorting arrays of objects, a sorting algorithm needs a way to compare elements to determine their order. However, the comparison rules vary between different classes.

The **Comparable** interface addresses this by providing a common method, *compareTo()*, which classes can implement to define their **natural ordering**.

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

Classes like Employee can implement Comparable <Employee> to enable sorting based on specific criteria such as employee IDs or salaries.

```
public class Employee implements Comparable
<Employee> {
    public int compareTo(Employee other) {
        return Double.compare(salary,
        other.salary);
    }
}
```

In this example, the compareTo() method compares employees based on their salaries.

3.3.2. COMPARATOR INTERFACE

The **Comparator** interface in Java provides a way to define **custom comparison logic for objects**, allowing for sorting based on criteria other than natural ordering.

The Comparator interface defines a single method *compare()*, which takes two objects of the same type and returns an integer value indicating their relative order.

```
public interface Comparator<T> {
    int compare(T first, T second);
}
```

Suppose we want to sort strings by their length rather than in dictionary order. We can achieve this by implementing a class that implements the Comparator<String> interface and defines the comparison logic accordingly.

```
class LengthComparator implements Comparator
<String> {
    public int compare(String first, String
    second) {
       return first.length() - second.length();
    }
}
```

In this example, the compare() method compares strings based on their lengths. A positive value indicates that the first string has a greater length than the second, while a negative value indicates the opposite.

To utilize the Comparator, we need to instantiate an object of the custom comparator class and use it to compare strings.

```
Comparator<String> comp = newLengthComparator();
   if (comp.compare(words[i], words[j]) > 0) {
      // words[i] comes after words[j] based on
         length
}
```

Here, the compare() method is invoked on the comparator object (comp), passing the strings to be compared (words[i] and words[j]).

3.3.3. RUNNABLE INTERFACE

The **Runnable** interface in Java serves as a building block for **executing tasks asynchronously**, particularly in scenarios where multi-threading is essential for optimizing resource utilization.

The Runnable interface defines a single method *run()*, which represents the code to be executed asynchronously.

Suppose we want to create a simple task that prints "Hello, World!" 1000 times. We can achieve this by implementing the Runnable interface in a custom class.

```
class HelloTask implements Runnable {
    public void run() {
        for (int i = 0; i < 1000; i++) {
           System.out.println(“Hello, World!”);
        }
    }
}
```

To execute the task represented by the HelloTask class in a separate thread, we instantiate a Thread object, passing the HelloTask object to its constructor, and then call the start() method on the Thread object.

```
Runnable task = new HelloTask();
Thread thread = new Thread(task);
thread.start();
```

3.3.4. USER INTERFACE CALLBACKS

In Java-based graphical user interface (GUI) libraries, such as JavaFX, interfaces are commonly used for specifying actions to be taken in response to user interactions, known as callbacks.

In JavaFX, the **EventHandler** interface is used for reporting events, such as button clicks, menu selections, or slider movements. It defines a single method *handle()* that takes an event object as a parameter.

```
public interface EventHandler<T> {
    void handle(T event);
}
```

Suppose we want to define an action to be performed when a user clicks a "Cancel" button. We can achieve this by implementing the EventHandler<ActionEvent> interface in a custom class.

```
class CancelAction implements
EventHandler<ActionEvent> {
    public void handle(ActionEvent event) {
        System.out.println("Oh noes!");
    }
}
```

To associate the callback with the "Cancel" button, we create an instance of the CancelAction class and set it as the event handler for the button.

```
Button cancelButton = new Button("Cancel");
cancelButton.setOnAction(new CancelAction());
```

3.4. LAMBDA EXPRESSIONS

A "**lambda expression**" is a block of code that you can pass around so it can be executed later, once or multiple times.

Java is an object-oriented language where (just about) everything is an object. There are no function types in Java. Instead, functions are expressed as objects, instances of classes that implement a particular interface. Lambda expressions give you a convenient syntax for creating such instances.

3.4.1. SYNTAX

In its simplest form, a lambda expression consists of **parameters**, an **arrow** (**->**), and a **body**.

```
(String first, String second) -> first.length()
- second.length()
```

If the computation within a lambda expression requires **multiple lines of code**, you enclose the body in curly braces **{}** and use explicit return statements.

```
(String first, String second) -> {
   int diff = first.length() - second.length();
   if (diff < 0) return -1;
   else if (diff > 0) return 1;
   else return 0;
}
```

If a lambda expression has **no parameters**, you still need to provide empty parentheses **()**.

```
Runnable task = () -> {
    for (int i = 0; i < 1000; i++) {
        doWork();
    }
}
```

If the parameter types of a lambda expression can be inferred from the context, you can omit them.

```
Comparator<Integer> comparator = (a, b) ->
                    a.compareTo(b);
```

3.4.2. FUNCTIONAL INTERFACES

Functional interfaces allow lambda expressions to seamlessly integrate with existing code that expects interfaces with a single abstract method.

A functional interface is an interface that specifies **only one abstract method**.

Consider the *Arrays.sort* method, which requires a *Comparator* object to specify the comparison logic for sorting. Since *Comparator* is a functional interface with a single method (*compare*), we can use a lambda expression to provide the comparison logic inline.

```
Arrays.sort(words,
  (first, second) -> first.length() - second.length());
```

This lambda expression is compatible with the *Comparator* interface, allowing it to be used as an argument to *Arrays.sort*.

When a lambda expression is passed to a method expecting a functional interface, the Java compiler automatically creates an instance of a class that implements that interface. The lambda expression's body becomes the implementation of the single abstract method defined in the interface.

Important Note

You cannot assign a lambda expression directly to a variable of type Object, which is the common supertype of all classes in Java. This restriction exists because **Object is a class**, **not a functional interface**. Lambda expressions can only be used where a functional interface is expected.

3.5. METHOD AND CONSTRUCTOR REFERENCES

Sometimes, there is already a method that carries out exactly the action that you'd like to pass on to some other code. There is special syntax for a **method reference** that is even shorter than a lambda expression calling the method. A similar shortcut exists for **constructors**.

3.5.1. METHOD REFERENCES

Method references allow you to **pass existing methods as arguments to other code**, such as functional interfaces, with a concise syntax.

Suppose you want to sort strings regardless of letter case. Instead of using a lambda expression, you can use a method reference:

```
Arrays.sort(strings,String::compareToIgnoreCase);
```

Here, *String::compareToIgnoreCase* is a method reference equivalent to the lambda expression *(x, y) -> x.compareToIgnoreCase(y)*.

The **::** operator separates the method name from the class or object name.

There are three variations of method references:

- **Class::instanceMethod**
  The first parameter becomes the receiver of the method, and any other parameters are passed to the method.
  Example: String::compareToIgnoreCase is equivalent to (x, y) -> x.compareToIgnoreCase(y).
- **Class::staticMethod**
  All parameters are passed to the static method.
  Example: Objects::isNull is equivalent to x -> Objects.isNull(x).
- **object::instanceMethod**
  The method is invoked on the given object, and the parameters are passed to the instance method.
  Example: System.out::println is equivalent to x -> System.out.println

3.5.1. CONSTRUCTOR REFERENCES

Constructor references are similar to method references, but instead of referring to methods, they refer to constructors. The syntax is **ClassName::new**, where ClassName is the name of the class.

Suppose you have a list of names and you want to create a list of Employee objects from these names without using a loop. You can achieve this using constructor references and streams:

```
List<String> names = ...;
Stream<Employee> stream =
names.stream().map(Employee::new);
```

Here, *Employee::new* is a constructor reference that creates *Employee* instances from *String* objects. When **map** is applied to the stream of names, each name is used as an argument to the *Employee* constructor.

Constructor references can also be used with **array types**. For example, **int[]::new** is a constructor reference for creating arrays of integers with a specified length. It is *equivalent to the lambda expression n -> new int[n]*.

```
Employee[] employees =
stream.toArray(Employee[]::new);
```

Here, *Employee[]::new* is a constructor reference that creates an array of Employee objects. The *toArray* method uses this constructor to obtain an array of the correct type, fills it with elements from the stream, and returns the array.

3.6. PROCESSING LAMBDA EXPRESSIONS

3.6.1. IMPLEMENTING DEFERRED EXECUTION

The essence of using lambda expressions lies in **deferred execution**, meaning the ability to execute code later rather than immediately.

Consider a simple example where you want to repeat an action n times. You can achieve this by passing the action and count to a repeat method:

```
repeat(10, () -> System.out.println("Hello,
                                    World!"));
```

To accept the lambda expression, you need to select a functional interface. In this case, you can use Runnable:

```
public static void repeat(int n, Runnable action){
    for (int i = 0; i < n; i++)
    action.run();
}
```

Here, the lambda expression's body is executed when *action.run*() is called within the repeat method.

3.6.2. CHOOSING A FUNCTIONAL INTERFACE

In most functional programming languages, function types are structural, meaning you specify a function's signature directly.

However, there are scenarios where you want to accept "any function" without specific semantics. In such cases, Java provides a range of generic function types. It's advisable to utilize one of these generic function types whenever possible, as it enhances code clarity and maintainability. (See Table 3–1 and 3–2)

3.6.3. IMPLEMENTING YOUR OWN FUNCTIONAL INTERFACES

Occasionally, standard functional interfaces may not suit your specific needs, prompting the creation of custom ones.

Suppose you're tasked with filling an image with color patterns, where the user provides a function that determines the color for each pixel. As there's no standard type for this mapping (int, int) -> Color, using BiFunction<Integer, Integer, Color> might incur autoboxing overhead.

In such cases, it's advisable to define a new functional interface:

```
@FunctionalInterface
public interface PixelFunction {
    Color apply(int x, int y);
}
```

Tagging functional interfaces with @FunctionalInterface is recommended. It ensures the interface has a single abstract method and provides clarity in documentation.

Now, you can implement a method:

```
BufferedImage createImage(int width, int
height, PixelFunction f) {
    BufferedImage image = new
    BufferedImage(width, height,
                    BufferedImage.TYPE_INT_RGB);
    for (int x = 0; x < width; x++)
        for (int y = 0; y < height; y++) {
            Color color = f.apply(x, y);
            image.setRGB(x, y, color.getRGB());
        }
    return image;
}
```

To utilize this method, provide a lambda expression that determines the color value for two integers:

```
BufferedImage frenchFlag = createImage(150,
100, (x, y) -> x < 50 ? Color.BLUE : x < 100 ?
Color.WHITE : Color.RED);
```

3.7. LAMBDA EXPRESSIONS AND VARIABLE SCOPE

3.7.1. SCOPE OF A LAMBDA EXPRESSION

The **scope of a lambda expression's body** is akin to that of a nested block, adhering to the same rules for name conflicts and shadowing. It's prohibited to declare a parameter or local variable within a lambda that shares the same name as a local variable in the enclosing scope.

```
int first = 0;
Comparator<String> comp = (first, second) ->
first.length() - second.length();
// Error: Variable first already defined
```

Within a method, having two local variables with identical names is not allowed, extending this restriction

to lambda expressions. Consequently, the this keyword within a lambda refers to the this parameter of the method creating the lambda, adhering to the "same scope" rule.

```
public class Application() {
    public void doWork() {
        Runnable runner = () -> {
           // ...
           System.out.println(this.toString());
           // ...
        };
    }
}
```

Here, *this.toString*() invokes the toString method of the Application object, not the Runnable instance. The **this keyword's behavior in a lambda is consistent** with its behavior in other parts of the enclosing method, as the lambda's scope is nested within *doWork*.

3.7.2. ACCESSING VARIABLES FROM THE ENCLOSING SCOPE

Lambda expressions often need to access variables from their enclosing method or class. Consider this example:

```
public static void repeatMessage(String text,
int count) {
    Runnable r = () -> {
        for (int i = 0; i < count; i++) {
            System.out.println(text);
        }
    };
    new Thread(r).start();
}
```

Here, the lambda expression accesses the text and count variables defined in the enclosing scope. When you call repeatMessage("Hello", 1000), it prints "Hello" 1000 times in a separate thread.

How do these variables retain their values when the lambda expression executes long after the enclosing method has returned?

Lambda expressions capture the values of variables from their enclosing scope. These captured values are stored internally within the lambda expression object. This mechanism ensures that the lambda expression has access to the necessary data even after the enclosing method has exited.

However, **lambda expressions can only capture variables that are effectively final**. An effectively final variable is one that is never modified after initialization or assignment. This prevents any ambiguity regarding the captured value.

For example, attempting to capture a non-final variable that changes its value within a loop would result in a compile-time error:

```
for (int i = 0; i < n; i++) {
    new Thread(() ->
    System.out.println(i)).start();
    // Error: cannot capture i
}
```

On the other hand, variables in an enhanced for loop are effectively final within each iteration:

```
for (String arg : args) {
    new Thread(() ->
    System.out.println(arg)).start();
    // OK to capture arg
}
```

It's important to note that **lambda expressions cannot modify the captured variables**. This restriction ensures thread safety and **avoids potential concurrency issues**. For instance, attempting to modify a captured variable inside a lambda expression results in a compilation error.

```
public static void repeatMessage(String text,
int count, int threads) {
    Runnable r = () -> {
        while (count > 0) {
            count--;
           // Error: Can't mutate captured
              variable
            System.out.println(text);
        }
    };
    for (int i = 0; i < threads; i++)
        new Thread(r).start();
}
```

3.8. HIGHER-ORDER FUNCTIONS

In functional programming languages, functions can be assigned to variables, passed as arguments to other functions, and returned as values from other functions.

Functions that either accept other functions as arguments or return functions as results are called **higher-order functions**.

Although Java is not purely functional, it supports the concept of higher-order functions through the use of **functional interfaces**.

3.8.1. METHODS THAT RETURN FUNCTIONS

Consider a scenario where you sometimes need to sort an array of strings in ascending order and other times in descending order. You can create a method that produces the appropriate comparator based on a specified direction:

```
public static Comparator<String>
compareInDirection(int direction) {
   return (x, y) -> direction * x.compareTo(y);
}
```

Here, the *compareInDirection* method returns a comparator that sorts strings based on the specified direction. For instance, *compareInDirection*(1) yields an ascending comparator, while *compareInDirection*(-1) produces a descending comparator.

You can then pass the resulting comparator to methods such as *Arrays.sort*, which expect a functional interface:

```
Arrays.sort(friends, compareInDirection(-1));
```

3.8.2. METHODS THAT MODIFY FUNCTIONS

We can extend the previous concept further by creating a method that modifies existing comparators.

Consider a method called *reverse*, which takes a comparator as input and returns a new comparator that sorts in the reverse order:

```
public static Comparator<String>
reverse(Comparator<String> comp) {
    return (x, y) -> comp.compare(x, y);
}
```

This method operates on functions, taking a comparator and returning a modified version of it. For instance, to obtain a case-insensitive descending order comparator, you can use:

```
reverse(String::compareToIgnoreCase)
```

Here, the *reverse* method allows for dynamic modification of comparators, enhancing flexibility and code reuse. It's worth noting that the Comparator interface provides a default method called **reversed**, which achieves the same functionality by producing the reverse of a given comparator.

3.8.3. COMPARATOR METHODS

The Comparator interface provides several useful **static methods that function as higher-order functions**, generating comparators for various sorting scenarios.

- **comparing:** This method takes a "key extractor" function, which maps objects of type T to a comparable type (e.g., String). It applies the function to the objects to be compared and performs the comparison based on the returned keys. For example, to sort an array of Person objects by name:

```
Arrays.sort(people,Comparator.comparing
(Person::getName));
```

- **thenComparing:** Chaining comparators allows breaking ties. For instance, to sort people by last name and then by first name if two individuals share the same last name:

```
Arrays.sort(people, Comparator
    .comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));
```

- **comparingInt**, **comparingLong**, **comparingDouble:** These variants avoid boxing of primitive values. For example, sorting by name length:

```
Arrays.sort(people, Comparator.comparingInt(p
-> p.getName().length()));
```

- **nullsFirst**, **nullsLast:** These methods adapt an existing comparator to handle null values without throwing exceptions. They rank null values as smaller or larger than regular values. For example, when sorting by potentially null middle names:

```
Arrays.sort(people,
comparing(Person::getMiddleName,
nullsFirst(naturalOrder())));
```

- **reverseOrder:** This method gives the reverse of the natural order. For instance, to sort in descending order:

```
Arrays.sort(people, reverseOrder());
```

3.9. LOCAL INNER CLASSES

Before the introduction of lambda expressions, Java provided a mechanism called **local inner classes** for **defining classes within methods**. These classes are useful for implementing interfaces or providing functionality that is only relevant within the method's scope.

3.9.1. LOCAL CLASSES

These classes are typically used when a class implements an interface and the caller of the method **only cares about the interface, not the specific class implementation**.

Consider a method *randomInts* that generates an infinite sequence of random integers within a specified range. Since *IntSequence* is an interface, the method needs to return an object of a class implementing that interface. However, the caller doesn't need to know the specifics of the implementing class, so it can be defined locally within the method:

```
private static Random generator = new Random();

public static IntSequence randomInts(int low,
int high) {
   class RandomSequence implements IntSequence{
        public int next() {
          return low + generator.nextInt(high -
          low + 1);
        }

        public boolean hasNext() {
            return true;
        }
   }

   return new RandomSequence();
}
```

Key points about local classes:

- They are **not declared as public or private** because they are not accessible outside the method.
- The **class name is hidden** within the method's scope.
- **Methods of the local class can access variables from the enclosing scope**, similar to lambda expressions.

In the example, the next method of RandomSequence captures the variables low, high, and generator from the enclosing scope.

If RandomSequence were a nested class, **explicit constructors would be required to pass and store these values as instance variables**.

3.9.2. ANONYMOUS CLASSES

In Java, you can create **anonymous classes**, which are **local classes without a name**. They are useful when you need to implement an interface or extend a class with a single-use instance.

Consider the *randomInts* method from the previous example, where we generated a sequence of random integers. Instead of defining a named local class, we can make the class anonymous:

```
public static IntSequence randomInts(int low,
int high) {
   return new IntSequence() {
      public int next() {
          return low + generator.nextInt(high -
           low + 1);
      }

      public boolean hasNext() {
          return true;
      }
   };
}
```

Key points about anonymous classes:

- They are **defined and instantiated in a single expression** using the **new keyword**.
- They **implement interfaces** or **extend classes** and **provide implementations for their methods** within the curly braces {}.
- The syntax **new Interface() { methods }** indicates the creation of an anonymous class that implements the specified interface or extends the specified class and provides the specified methods.
- The parentheses () following the new keyword indicate the **arguments to be passed to the constructor of the anonymous class**. If there are no arguments, an empty pair of parentheses is used to invoke the default constructor.

Anonymous classes are useful for situations where a class is only needed once and does not require a separate, named definition.

INHERITANCE AND REFLECTION

Inheritance is the process of creating new classes that are built on existing classes. When you inherit from an existing class, you reuse (or inherit) its methods, and you can add new methods and fields.

Note: Instance variables and static variables are collectively called **fields**. The fields, methods, and nested classes/interfaces inside a class are collectively called its **members**.

Reflection, the ability to find out more about classes and their members in a running program. Reflection is a powerful feature, but it is undeniably complex.

4.1. EXTENDING A CLASS

Inheritance is exemplified through the **extension of existing classes to create new ones**.

Returning to our Employee class, let's consider a scenario where managers have distinct characteristics and behaviors.

4.1.1. SUPER- AND SUBCLASSES

We introduce a new class, Manager, which **inherits some functionalities** from the Employee class while **specifying the differences** for managers.

```
public class Manager extends Employee {
  // Additional fields and methods for managers
}
```

The **extends** keyword denotes the creation of a new class derived from an existing one.

In this context, the existing class is referred to as the **superclass**, while the new class is termed the **subclass**. Despite the terminology, **subclasses possess more functionalities than their superclasses**.

4.1.2. DEFINING AND INHERITING SUBCLASS METHODS

Our Manager class has a new instance variable to store the bonus and a new method to set it:

```
public class Manager extends Employee {
    private double bonus;

    public void setBonus(double bonus) {
        this.bonus = bonus;
    }
}
```

When instantiating a Manager object, both subclass-specific methods and inherited methods from the superclass, such as *raiseSalary*, can be invoked.

```
Manager boss = new Manager(...);
boss.setBonus(10000);  // Defined in subclass
boss.raiseSalary(5);   // Inherited from
                          superclass
```

4.1.3. METHOD OVERRIDING

Method overriding is a mechanism in object-oriented programming where **a subclass provides a specific implementation of a method that is already defined in its superclass**. This allows for customization of behavior for subclasses while maintaining a common interface defined by the superclass.

In our example, let's consider the *getSalary* method in the Manager class. The superclass method, inherited from Employee, may not be sufficient for managers, as they may receive additional bonuses on top of their base salary. Thus, we override the getSalary method in the Manager class to include the bonus:

```
public class Manager extends Employee {
    // Other class members and methods...

    @Override
    public double getSalary() {
        // Overrides superclass method to
           include bonus
        return super.getSalary() + bonus;
    }
}
```

Here, the **super** keyword is used to invoke the superclass method, *getSalary*(), to retrieve the base salary. The bonus is then added to this value to compute the total salary for the manager.

A subclass method cannot directly access private instance variables of the superclass. This is why we invoke the public getSalary method of Employee within Manager.

When overriding a method, it's common practice to **match the parameter types exactly** with those of the superclass method. By using the **@Override** annotation, we can ensure that we're correctly overriding a superclass method. This annotation helps catch errors where a new method is accidentally defined instead of overriding a superclass method (because if the parameter does not match, you are not overriding but rather defining a new method).

Additionally, it's **permissible to change the return type to a subtype** when overriding a method, known as covariant return types. This allows for greater flexibility in method overriding.

Important Note

The **visibility of the subclass method must be at least as visible as the superclass method**. If the superclass method is public, then the subclass method must also be declared public. Omitting the public modifier for the subclass method can lead to compilation errors due to weaker access privilege.

4.1.4. SUBCLASS CONSTRUCTION

When creating a subclass, it's often necessary to provide **constructors to initialize its state**.

In our case, let's define a constructor for the Manager class. Since the Manager constructor cannot directly access the private instance variables of the Employee class, we must initialize them through a **superclass constructor**.

```
public Manager(String name, double salary) {
    super(name, salary);    // Calls superclass
                               constructor
    bonus = 0; // Initializes bonus
}
```

Here, the **super** keyword indicates a call to the constructor of the Employee superclass with name and salary as arguments.

It's crucial that the **superclass constructor call is the first statement in the constructor for the subclas**s.

If you omit the superclass constructor call, the superclass must have a no-argument constructor, which is implicitly called. This ensures that the superclass is properly initialized before the subclass constructor executes.

4.1.5. SUPERCLASS ASSIGNMENTS

In Java, it's legal to **assign an object from a subclass to a variable whose type is a superclass**. For example:

```
Manager boss = new Manager(...);
Employee empl = boss; // OK to assign to
                         superclass variable
```

When invoking a method on the superclass variable, such as *empl.getSalary*(), **the method from the actual class of the object is invoked**.
This is known as **dynamic method lookup**. This approach allows for writing code that works for all employees, whether they are managers, janitors, or other types of employees.

For instance, you can create an array of Employee objects and assign various subclasses to its elements:

```
Employee[] staff = new Employee[...];
staff[0] = new Employee(...);
staff[1] = new Manager(...); // OK to assign to
                              superclass variable
staff[2] = new Janitor(…);
// Other assignments...
double sum = 0;
for (Employee empl : staff)
    sum += empl.getSalary();
```

Thanks to dynamic method lookup, invoking *empl.getSalary*() within the loop will correctly call the *getSalary* method of the object's actual class, whether it's Employee, Manager, or Janitor.

Important Note

While superclass assignment also works for arrays, it can lead to type errors at runtime if not used carefully. For instance:

```
Manager[] bosses = new Manager[10];
Employee[] empls = bosses; // Legal in Java
empls[0] = new Employee(...); // Runtime error
```

Here, even though the assignment of bosses to *empls* is legal, attempting to store an *Employee* object in a *Manager* array leads to a runtime *ArrayStoreException*. This is because empls and bosses reference the same array, which is typed as *Manager*[].

4.1.6. CASTS

A variable of type *Employee* can reference objects of various subclasses, such as *Manager*. However, there's a limitation: **you can only invoke methods that belong to the superclass**. For instance:

```
Employee empl = new Manager(...);
empl.setBonus(10000); // Compile-time error
```

Even though this call could succeed at runtime, it's a *compile-time error* because the **compiler ensures that you only invoke methods that exist for the receiver type**. Here, *empl* is of type *Employee*, which doesn't have a *setBonus* method.

To work around this limitation, you can use the **instanceof** operator and a **cast** to convert a superclass reference to a subclass:

```
if (empl instanceof Manager) {
    Manager mgr = (Manager) empl;
    mgr.setBonus(10000);
}
```

This conditional check ensures that *empl* is indeed an instance of *Manager* before casting it to Manager. This way, you can safely invoke setBonus.

4.1.7. FINAL METHODS AND CLASSES

When you declare a method as **final**, **no subclass can override it**. For example:

```
public class Employee {
    ...
    public final String getName() {
        return name;
    }
}
```

A common use case for final methods is seen in the *getClass* method of the *Object class*. This method **cannot be overridden**, ensuring that objects cannot misrepresent their class.

Occasionally, you may want to prevent subclassing of your class entirely. You can achieve this by marking the class as final:

```
public final class Executive extends Manager {
    ...
}
```

4.1.8. ABSTRACT METHODS AND CLASSES

In Java, a class can define a method without an implementation, which forces subclasses to implement it. Such a method, along with the class containing it, is termed abstract and must be marked with the **abstract**

modifier. Abstract classes are commonly used for very general classes.

For instance:

```
public abstract class Person {
    private String name;
    public Person(String name) {
       this.name = name;
    }
    public final String getName() {
       return name;
    }
    public abstract int getId();
}
```

Here, any class extending Person must either provide an implementation for the *getId* method or be declared as abstract itself.

Note that **abstract classes can have non-abstract methods as well**, such as the getName method in the example.

Unlike interfaces, **abstract classes can have instance variables and constructors**. However, **it's not possible to directly instantiate an abstract class**. For example, the following would result in a compile-time error:

```
Person p = new Person("Fred"); // Error
```

However, you can have a variable whose type is an abstract class, provided it contains a reference to an object of a concrete subclass. For example:

```
Person p = new Student("Fred", 1729);
// OK, a concrete subclass
```

4.1.9. PROTECTED ACCESS

In Java, there are situations where you want to **restrict a method to subclasses only** or **allow subclass methods to access an instance variable of a superclass**. For this purpose, you can declare a class feature as **protected**.

However, note that in Java, protected grants package-level access and only protects access from other packages. For instance, if the superclass *Employee* declares the instance variable salary as protected:

```
package com.horstmann.employees;
public class Employee {
    protected double salary;
    ...
}
```

All classes in the same package as Employee can access this field. Consider a subclass in a different package:

```
package com.horstmann.managers;
import com.horstmann.employees.Employee;
public class Manager extends Employee {
    ...
    public double getSalary() {
        return salary + bonus;
    // OK to access protected salary variable
    }
}
```

In this case, the *Manager* class methods can access the salary variable of *Manager* objects only, not of other *Employee* objects. This restriction prevents the abuse of the protected mechanism by forming subclasses solely to gain access to protected features.

4.1.10. ANONYMOUS SUBCLASSES

Just as you can have an anonymous class that implements an interface, you can also have an **anonymous class that extends a superclass**. This feature can be particularly useful for debugging purposes.

For example:

```
ArrayList<String> names = newArrayList<String>
(100) {
   public void add(int index, String element) {
      super.add(index, element);
      System.out.printf("Adding %s at %d\n",
                        element, index);
    }
};
```

Here, an anonymous subclass of ArrayList<String> is created, overriding the add method. The instance is constructed with an initial capacity of 100.

There's also a technique called **double brace initialization**, which utilizes inner class syntax in a peculiar way.

For instance, if you want to construct an array list and pass it to a method without needing it again:

```
invite(new ArrayList<String>()
{{ add("Harry"); add("Sally"); }});
```

Note the double braces. The outer braces create an anonymous subclass of ArrayList<String>, while the inner braces form an initialization block.

4.1.11. INHERITANCE AND DEFAULT METHODS

In scenarios where a class extends another class and implements an interface, and both have a method with the same name, the **superclass implementation always takes precedence over the interface implementation**. Therefore, there's no need for the subclass to resolve the conflict in this situation.

For example:

```
public interface Named {
    default String getName() { return ""; }
}
public class Person {
    public String getName() { return name; }
}
public class Student extends Person implements
Named {
    // No need to resolve conflict between
       Person and Named
}
```

This rule, known as the **"classes win" rule**.

4.1.12. METHOD EXPRESSIONS WITH 'SUPER'

In Java, method expressions can take the form **object::instanceMethod**, where object is an object reference and instanceMethod is the method to be invoked on that object.

However, instead of an object reference, it's also valid to use **super** to refer to the superclass.

For example:

```
public class Worker {
    public void work() {
        for (int i = 0; i < 100; i++)
            System.out.println("Working");
    }
}
public class ConcurrentWorker extends Worker {
    public void work() {
        Thread t = new Thread(super::work);
        t.start();
    }
}
```

In this example, the *ConcurrentWorker* class extends the *Worker class*. Inside the work method of *ConcurrentWorker*, a new thread is created, and the *super*::*work* method expression is used to invoke the work method of the superclass (*Worker*). This means that the work method of the superclass will be executed in a separate thread when work is called on an instance of *ConcurrentWorker*.

4.2. OBJECT: THE COSMIC SUPERCLASS

In Java, every class directly or indirectly extends the Object class. When a class has no explicit superclass specified, it implicitly extends Object. For instance, declaring a class as:

```
public class Employee { ... }
```

is equivalent to

```
public class Employee extends Object { ... }
```

The Object class defines methods that are applicable to any Java object. These methods are inherited by all classes. Arrays, which are also classes in Java, can be converted to a reference of type Object, including arrays of primitive types.

4.2.1. THE toString METHOD

The **toString** method in the Object class is crucial as it provides a string representation of an object. For instance, the toString method of the Point class typically returns a string in the format *java.awt.Point[x=10,y=20]*, where the class name is followed by the instance variables enclosed in square brackets.

Here's an example implementation of the *toString* method for the *Employee* class:

```
public String toString() {
  return getClass().getName() + "[name=" + name
  + ",salary=" + salary + "]";
}
```

Using *getClass*().*getName*() instead of hardwiring the class name ensures that the method works correctly for subclasses as well.

In a subclass, you can call **super.toString()** to include the string representation of the superclass, followed by additional instance variables specific to the subclass:

```
public class Manager extends Employee {
    // ...
    public String toString() {
        return super.toString() + "[bonus=" +
        bonus + "]";
    }
}
```

When an object is concatenated with a string, the Java compiler automatically invokes the toString method on the object. For example:

```
Point p = new Point(10, 20);
String message = "The current position is " + p;
// Automatically calls p.toString()
```

Instead of explicitly calling x.toString(), you can concatenate x with an empty string ("" + x). This expression even works if x is null or a primitive type value.

If a class does not override the toString method, the default implementation from the Object class is used. This default implementation prints the class name followed by the hash code of the object.

```
System.out.println(System.out); // Prints
something like java.io.PrintStream@2f6684
```

Arrays inherit the toString method from Object, but the default behavior prints an archaic format indicating the array type. For example:

```
int[] primes = { 2, 3, 5, 7, 11, 13 };
System.out.println(primes); // Prints something
                                like [I@1a46e30
```

To get a meaningful string representation of an array, use Arrays.toString(primes) instead:

```
System.out.println(Arrays.toString(primes));
// Prints [2, 3, 5, 7, 11, 13]
```

For multidimensional arrays, use **Arrays.deepToString** for correct printing.

4.2.2. THE equals METHOD

The **equals** method in Java is used to test whether one object is considered equal to another. By default, the equals method in the Object class determines whether **two object references are identical**, which means they **refer to the same memory location**. However, this behavior might not always be suitable, especially for comparing the contents of objects.

Override the equals method only for state-based equality testing, in which two objects are considered equal when they have the same contents. For example, the String class overrides equals to check whether two strings consist of the same characters.

For instance, let's say we have a class Item with instance variables description and price. We want to consider two Item objects equal if their descriptions and prices match.

Here's how we can implement the equals method for this class:

```
public boolean equals(Object otherObject) {
    if (this == otherObject) return true;
       // Quick identity check
    if (otherObject == null) return false;
      // Must return false if parameter is null
    if (getClass() != otherObject.getClass())
    return false; // Check class compatibility

    Item other = (Item) otherObject;
    return Objects.equals(description,
    other.description) && price == other.price;
}
```

It's important to follow some routine steps when implementing the equals method:

- Consider adding a quick **identity check at the beginning**.
- Always **return false when comparing against null**.
- Perform a **type check** before casting the parameter to the actual type.
- Finally, **compare the instance variables to determine equality**.

When you define the equals method for a subclass, first **call equals on the superclass**. If that test doesn't pass, the objects can't be equal. If the instance variables of the superclass are equal, then you are ready to compare the instance variables of the subclass.

For instance, let's consider a class *DiscountedItem* which extends the Item class and adds a discount field.

```
public class DiscountedItem extends Item {
    private double discount;
    ...
    public boolean equals(Object otherObject) {
       if (!super.equals(otherObject)) return
           false;
       DiscountedItem other = (DiscountedItem)
           otherObject;
       return discount == other.discount;
}
    public int hashCode() { ... }
   }
```

Note that the getClass test in the superclass (Item) fails if otherObject is not a DiscountedItem. This is because getClass returns the runtime class of an object, and if otherObject is not of type DiscountedItem, it means it's not an instance of the subclass.

4.2.3. THE hashCode METHOD

A hash code is an integer value derived from an object. Hash codes are "scrambled" - meaning, if two objects are unequal, their hash codes should likely be different.

The String class, for example, calculates its hash code using a specific algorithm. It iterates through each character of the string, multiplying the current hash code by 31 and adding the ASCII value of the character.

```
int hash = 0;
   for (int i = 0; i < length(); i++)
       hash = 31 * hash + charAt(i);
```

Compatibility between the hashCode and equals methods is essential. **If two objects are equal** according to the equals method, **their hash codes must be equal** as well. Conversely, if two objects have different hash codes, they must be unequal.

Important Note

When you override the equals method, it's imperative to override the hashCode method as well to maintain compatibility. Failure to do so can lead to issues when inserting objects into hash-based collections like HashSet or HashMap.

A common practice for implementing hashCode is to **combine the hash codes of the object's instance variables**. For example, a class Item might compute its hash code by hashing its description and price fields using Objects.hash.

If your class contains array instance variables, you should compute their hash codes separately using Arrays.hashCode and then combine them with other instance variables using Objects.hash.

It's worth noting that in Java interfaces, you cannot provide default methods for toString, equals, or hashCode. This is because, according to the "classes win" rule in Java's inheritance mechanism, such default methods would always be overridden by the implementations in the implementing classes.

4.2.4. CLONING OBJECTS

The purpose of the clone method is to **produce a distinct object with the same state as the original**. This means that if you modify one of the objects, the other remains unchanged.

For instance, consider the scenario where you have an Employee object fred, and you want to create a clone called cloneOfFred. If you raise the salary of cloneOfFred, the salary of fred remains unaffected.

By default, the clone method is declared as **protected** in the *Object* class. **If you want users of your class to be able to clone instances, you must override this method in your class**.

The default behavior of Object.clone creates a **shallow copy**. This means it only copies the instance variables from the original object to the clone. While this is fine for primitive or immutable variables, it can lead to issues when dealing with mutable variables.

For example, consider a class Message representing email messages. It contains a list of recipients among its instance variables. If you create a shallow copy of a Message object, both the original and the clone will share the same recipients list. Any modification to this list

in one object will be reflected in the other, which is not desirable behavior.

To address this, our class should override the clone method to make a **deep copy**. In a deep copy, not only are the instance variables copied, but any objects referenced by those variables are also copied, creating **separate instances**.

When implementing a class, you need to decide whether to:

- **provide a clone method**,
- **use the inherited clone method**,
- or **implement a custom clone method** for making a deep copy.

For the first option, if you choose not to provide a clone method, simply do nothing. Your class will inherit the clone method, but it will remain inaccessible to users since it is declared as protected.

For the second option, if you decide to use the inherited clone method, your class must implement the Cloneable interface. This interface acts as a marker interface without any methods. The Object.clone method checks if the class implements Cloneable before making a shallow copy and throws a CloneNotSupportedException if it doesn't.

To use the inherited clone method, you should raise its scope from protected to public and change the return type to match that of your class.

For example:

```
public class Employee implements Cloneable {
   ...
   public Employee clone() throws
   CloneNotSupportedException {
           return (Employee) super.clone();
   }
}
```

The cast (Employee) is necessary because the return type of Object.clone is Object.

4.3. ENUMERATIONS

Enumerations provide a way to define a **fixed set of named constant values**. For example, you can define an enumeration for sizes like SMALL, MEDIUM, LARGE, and EXTRA_LARGE as follows:

```
public enum Size {SMALL, MEDIUM, LARGE, EXTRA_LARGE};
```

4.3.1. THE toString METHOD

Enumerations offer various methods to work with the defined constants.

- Firstly, since each instance of an enumerated type has a fixed set of values, you can directly compare them using the **== operator**. The equals method is unnecessary in this context.
- Similarly, the **toString method** is automatically provided for enumerations, returning the name of the enumerated constant. For instance, `Size.SMALL.toString()` would return "SMALL".
- To obtain an instance of an enumeration from its string representation, you can use the **valueOf method**. For example:

```
Size notMySize = Size.valueOf("SMALL");
```

  This sets notMySize to Size.SMALL. If the specified name does not match any instance, valueOf throws an exception.

- You can retrieve all instances of an enumeration using the **values method**, which returns an array of all defined constants. For instance:

```
Size[] allValues = Size.values();
```

- The **ordinal method** of an enumeration returns the position of an instance in the declaration, starting from zero. For instance, `Size.MEDIUM.ordinal()` returns 1.

Every enumerated type **automatically implements the Comparable<E> interface**, allowing comparisons only against its own objects. The comparison is based on ordinal values, facilitating sorting and ordering of enumeration constants.

4.3.2. CONSTRUCTORS, METHODS, AND FIELDS

Enumerated types can have **constructors**, **methods**, and **fields** just like regular classes.

Each instance of the enumeration is guaranteed to be constructed exactly once. However, the **constructor of an enumeration is always implicitly private**, so explicitly declaring it as private is unnecessary.

For example, consider an enumeration representing clothing sizes with corresponding abbreviations:

```
public enum Size {
    SMALL("S"), MEDIUM("M"), LARGE("L"),
    EXTRA_LARGE("XL");

    private String abbreviation;

    Size(String abbreviation) {
        this.abbreviation = abbreviation;
    }

    public String getAbbreviation() {
        return abbreviation;
    }
}
```

4.3.3. BODIES OF INSTANCES

You can **add methods to individual enum instances by using enum bodies**. These methods must override methods defined in the enumeration. For instance, you could implement a calculator using an enumeration representing arithmetic operations:

```
public enum Operation {
```

```
   ADD {
      public int eval(int arg1, int arg2) {
         return arg1 + arg2;
       }
    },
    SUBTRACT {
       public int eval(int arg1, int arg2) {
           return arg1 - arg2;
        }
    },
    MULTIPLY {
       public int eval(int arg1, int arg2) {
          return arg1 * arg2;
        }
    },
    DIVIDE {
       public int eval(int arg1, int arg2) {
          return arg1 / arg2;
        }
    };
    public abstract int eval(int arg1, int
    arg2);
}
```

In this example, each operation constant overrides the eval method to perform the respective arithmetic operation.

Note

Each constant in an enumeration technically belongs to an anonymous subclass of the enumeration. Therefore, anything that can be placed into an anonymous subclass body can also be added into the body of a member within the enumeration.

4.3.4. STATIC MEMBERS

It is legal for an enumeration to have static members. However, you have to be careful with construction order. The **enumerated constants are constructed before the static members**, so you **cannot refer to any static members in an enumeration constructor**. For example, the following would be illegal:

```
public enum Modifier {
   PUBLIC, PRIVATE, PROTECTED, STATIC, FINAL,
   ABSTRACT ;
      private static int maskBit = 1;
      private int mask;
      public Modifier() {
           mask = maskBit;
  // Error—cannot access static variable in
     constructor
           maskBit *= 2;
       }
...
}
```

To resolve this issue, initialize static variables in a static initializer block, where static members can be accessed safely.

```
public enum Modifier {
    PUBLIC, PRIVATE, PROTECTED, STATIC, FINAL,
    ABSTRACT;
    private int mask;
    static {
        int maskBit = 1;
        for (Modifier m : Modifier.values()) {
            m.mask = maskBit;
            maskBit *= 2;
        }
    }
}
```

Note

Enumerated types can be **nested inside classes**. Such nested enumerations are implicitly static nested classes—that is, their methods cannot reference instance variables of the enclosing class.

4.3.5. SWITCHING ON AN ENUMERATION

You can use enumeration constants in a switch statement.

```
enum Operation { ADD, SUBTRACT, MULTIPLY, DIVIDE };
public static int eval(Operation op, int arg1,
int arg2) {
   int result = 0;
   switch (op) {
      case ADD: result = arg1 + arg2; break;
      case SUBTRACT: result = arg1 - arg2;
                     break;
      case MULTIPLY: result = arg1 * arg2;
                     break;
      case DIVIDE: result = arg1 / arg2; break;
    }
    return result;
}
```

You use **ADD**, **not Operation.ADD**, inside the switch statement—the type is inferred from the type of the expression on which the switch is computed.

4.4. RUNTIME TYPE INFORMATION AND RESOURCES

In Java, you can find out at runtime to which class a given object belongs. This is sometimes useful, for example in the implementation of the equals and toString methods. Moreover, you can find out how the class was loaded and load its associated data, called resources.

4.4.1. THE Class CLASS

Suppose you have a variable of type Object, filled with some object reference, and you want to know more about the object, such as to which class it belongs.

The **getClass** method yields an object of a class called **Class**.

```
   Object obj = ...;
   Class<?> cl = obj.getClass();
```

Once you have a Class object, you can retrieve information about the class, such as its name.

```
System.out.println("This object is an instance
of " + cl.getName());
```

Alternatively, you can obtain a Class object using the **Class.forName** method, passing the fully qualified class name as a string.

```
String className = "java.util.Scanner";
Class<?> cl = Class.forName(className);
// Describes the java.util.Scanner class
```

If you know the class you want in advance, you can use a **class literal** instead.

```
Class<?> cl = java.util.Scanner.class;
```

You can also use the **.class** suffix to get information about other types, including arrays, interfaces, primitive types, and void.

```
Class<?> cl2 = String[].class;
// Describes the array type String[]
Class<?> cl3 = Runnable.class;
// Describes  the Runnable interface
Class<?> cl4 = int.class;
// Describes the int type
Class<?> cl5 = void.class;
// Describes the void type
```

When working with array types, note that **getName** returns a strange notation, but you can use **getCanonicalName** to get a more readable representation.

```
String arrayName = String[].class.getName();
```

Returns "[Ljava.lang.String;"

```
String canonicalArrayName =
String[].class.getCanonicalName();
```

Returns "java.lang.String[]"

The virtual machine maintains a unique Class object for each type, allowing you to compare class objects using the **== operator**.

```
if (other.getClass() == Employee.class) {
    // Do something
}
```

4.4.2. LOADING RESOURCES

One useful service of the Class class is to locate resources that your program may need, such as configuration files or images. If you place a resource into the same directory as the class file, you can open an input stream to the file like this:

```
InputStream stream =
MyClass.class.getResourceAsStream("config.txt");
Scanner in = new Scanner(stream);
```

4.4.3. CLASS LOADERS

Java uses class loaders to **load class files into the virtual machine dynamically**. Three main class loaders are involved when executing a Java program:

- The **bootstrap class loader** loads Java library classes.
- The **extension class loader** loads standard extensions from a designated directory.
- The **system class loader** loads application classes from directories and JAR files on the classpath.

The bootstrap class loader, which loads Java library classes, is not represented by a ClassLoader object. The extension and system class loaders are typically instances of URLClassLoader.

You can obtain the class path of your program using the *getURLs* method, which returns an array of URL objects representing the directories and JAR files on the classpath.

To load classes from directories or JAR files not on the classpath, you can create your own URLClassLoader instance. This is commonly done for loading plugins.

4.4.4. THE CONTEXT CLASS LOADERS

In Java, most of the time, the class loading process is transparent, with classes being loaded as needed by other classes. However, issues can arise when a method loads classes dynamically, especially if it's called from a class loaded with a different class loader. This situation, known as **classloader inversion**, can cause problems.

Consider a scenario where:

- You have a utility class, Util, loaded by the system class loader, with a method createInstance that dynamically loads a class.
- A plugin, loaded by another class loader, calls Util.createInstance("com.mycompany.plugins.MyClass") to instantiate a class in the plugin JAR.

The problem arises because *Util.createInstance* uses its own class loader, which doesn't have visibility into the plugin JAR.

One solution is to **pass the class loader to the utility method and then to the Class.forName method**.

```
public class Util {
    public Object createInstance(String
    className, ClassLoader loader) {
        Class<?> cl = Class.forName(className,
        true, loader);
        // Instantiate the class
        // ...
    }
    // Other methods
}
```

Alternatively, you can **utilize the context class loader of the current thread**. By default, the main thread's context class loader is the system class loader, but you can set it to any class loader.

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

Then, the utility method can retrieve the context class loader to load the class dynamically.

```
public class Util {
   public Object createInstance(String
   className) {
      Thread t = Thread.currentThread();
      ClassLoader loader =
                     t.getContextClassLoader();
      Class<?> cl = Class.forName(className,
      true, loader);
        // Instantiate the class
        // ...
    }
    // Other methods
}
```

When invoking a method of a plugin class, the application should set the context class loader to the plugin class loader and restore the previous setting afterward.

4.4.5. SERVICE LOADERS

In Java, the **ServiceLoader** class simplifies the process of loading plugins that adhere to a common interface.

Here's how it works:

- First, define an interface or superclass with the methods that each instance of the service should provide.
  For example, a Cipher interface might include methods for encryption, decryption, and determining strength.
- Next, the service provider creates one or more classes that implement this interface.
  For instance, a CaesarCipher class might implement the Cipher interface with its own encryption and decryption logic.
  Implementing classes can reside in any package and must have a no-argument constructor.
  To facilitate service loading, add the names of the implementing classes to a UTF-8 encoded text file in the *META-INF/services* directory. This file should be named after the fully qualified name of the service interface. For example, the file *META-INF/services/com.corejava.crypt.Cipher* would contain the name of the implementing class, such as *com.corejava.crypt.impl.CaesarCipher*.
- Once the preparation is complete, initialize a **ServiceLoader instance** in the program using the **ServiceLoader.load() method**. This initialization should occur only once in the program.
  The ServiceLoader instance provides an iterator through all provided implementations of the service. You can use an enhanced for loop to traverse them and pick an appropriate object to carry out the service based on certain criteria, such as strength.

Here's an example method that retrieves a cipher with a minimum strength:

```
public static Cipher getCipher(int minStrength) {
    for (Cipher cipher : cipherLoader) {
        if (cipher.strength() >= minStrength) {
            return cipher;
        }
    }
    return null;
}
```

In this method, cipherLoader is the ServiceLoader instance previously initialized, and strength() is a method defined in the Cipher interface to determine the strength of the cipher implementation.

4.5. REFLECTION

Reflection allows a program to **inspect the contents of arbitrary objects at runtime** and to **invoke arbitrary methods on them**. This capability is useful for implementing tools such as object-relational mappers or GUI builders.

4.5.1. ENUMERATING CLASS MEMBERS

In reflection, classes like **Field**, **Method**, and **Constructor** in the **java.lang.reflect package** describe the fields, methods, and constructors of a class, respectively.

All three classes have a method called **getName** that returns the *name of the member*.

The **Field** class has a method **getType** that *returns an object*, again *of type Class*, that describes the field type.

The **Method** and **Constructor** classes have *methods to report the types of the parameters*, and the Method class also *reports the return type*.

All three of these classes also have a method called **getModifiers** that returns an integer, with various bits turned on and off, that *describes the modifiers used (such as public or static)*. You can use static methods such as **Modifier.isPublic** and **Modifier.isStatic** to *analyze the integer that getModifiers returns*. The **Modifier.toString** returns a string of all modifiers.

The **getFields**, **getMethods**, and **getConstructors** methods of the *Class class* return arrays of the public fields, methods, and constructors that the class supports; this includes *public inherited members*. The **getDeclaredFields**, **getDeclaredMethods**, and **getDeclaredConstructors** methods return arrays consisting of all fields, methods, and constructors that are declared in the class. This *includes private, package, and protected members, but not members of superclasses.*

The **getParameters** method of the *Executable class*, the common superclass of Method and Constructor, returns an array of Parameter objects describing the method parameters.

4.5.2. INSPECTING OBJECTS

These objects can do more: They can also peek into objects and **retrieve field values**.

For example, here is how to enumerate the contents of all fields of an object:

```
Object obj = ...;
for (Field f : obj.getClass().getDeclaredFields()) {
   f.setAccessible(true);
   Object value = f.get(obj);
   System.out.println(f.getName() + ":" + value);
}
```

The key is the **get method** that reads the field value. If the field value is a primitive type value, **a wrapper object is returned**; in that case you can also call one of the methods getInt, getDouble, and so on.

In the same way, you can **set a field**. This code will give a raise to obj, no matter to which class it belongs, provided that it has a salary field of type double or Double. (If not, an exception will occur.)

```
Field f = obj.getDeclaredField("salary");
f.setAccessible(true);
double value = f.getDouble(obj);
f.setDouble(obj, value * 1.1);
```

4.5.3. INVOKING METHODS

Just like a Field object can be used to read and write fields of an object, a Method object can **invoke the given method on an object**.

```
Method m = ...;
Object result = m.invoke(obj, arg1, arg2, …);
```

If the method is static, supply null for the initial argument.

To obtain a method, you can search through the array returned by the *getMethods* or *getDeclaredMethods* method or you can call *getMethod* and supply the parameter types.

For example, to get the *setName(String)* method on a Person object:

```
Person p = ...;
Method m = p.getClass().getMethod("setName",
           String.class);
p.invoke(obj, "********");
```

4.5.4. CONSTRUCTING OBJECTS

To construct an object with the no-argument constructor, simply call **newInstance** on a Class object:

```
Class<?> cl = ...;
Object obj = cl.newInstance();
```

To invoke another constructor, you first need to find the Constructor object and then call its newInstance method. For example, suppose you know that a class has a public constructor whose parameter is an int. Then you can construct a new instance like this:

```
Constructor constr = cl.getConstructor(int.class);
Object obj = constr.newInstance(42);
```

EXCEPTIONS, ASSERTIONS, AND LOGGING

5.1. EXCEPTION HANDLING

Checking for errors and propagating error codes up the chain of method calls often led to errors going undetected. Java addresses this issue with exception handling, allowing a method to **signal a serious problem by "throwing" an exception**, which can be caught and handled by a method further up the call chain.

5.1.1. THROWING EXCEPTIONS

A method may encounter scenarios where it cannot carry out the intended task, such as when a required resource is missing or when inconsistent parameters are provided. In such cases, it is advisable to throw an exception.

For example, consider a method *randInt* that generates a random integer between two bounds:

```
private static Random generator = new Random();
public static int randInt(int low, int high) {
   return low + (int) (generator.nextDouble() *
   (high - low + 1));
}
```

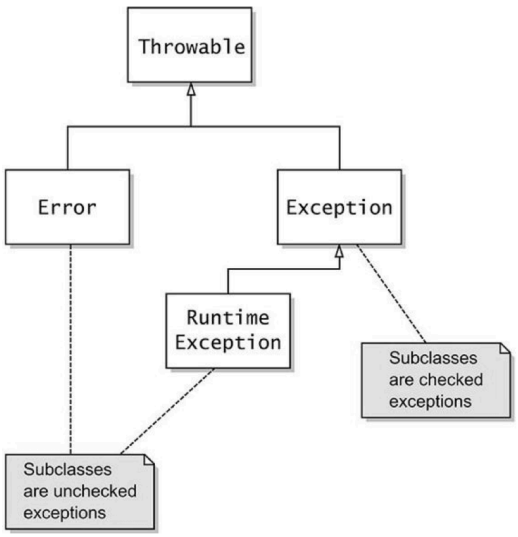
If someone calls *randInt(10, 5)*, indicating inconsistent bounds, attempting to fix this internally may not be ideal. Instead, throwing an **appropriate exception** is preferred:

```
if (low > high)
   throw new IllegalArgumentException(
   String.format("low should be <= high but low
            is %d and high is %d", low, high));
```

Here, the throw statement is used to throw an object of the **IllegalArgumentException class**, constructed with a debugging message. Upon executing a throw statement, the **normal flow of execution is immediately interrupted**, and **control is transferred to a handler**.

5.1.2. THE EXCEPTION HIERARCHY

All exceptions are **subclasses of the class Throwable**.



Subclasses of Error are exceptions that are thrown when something exceptional happens that the program cannot be expected to handle, such as memory exhaustion. There is not much you can do about errors other than giving a message to the user that things have gone very wrong.

Programmer-reported exceptions fall into two categories:

- **Unchecked exceptions:** Subclasses of RuntimeException. These exceptions are not checked during compilation. Indicate logic errors caused by programmers and are not explicitly checked. A common example is NullPointerException, which occurs when attempting to dereference a null object.
- **Checked exceptions:** All other exceptions that are not subclasses of RuntimeException. These are used for situations where failure is anticipated, such as input/output operations where files may be damaged or network connections may fail. For example, a FileNotFoundException is thrown when an expected file is not found.

When creating custom exception classes, it's advisable to **extend Exception, RuntimeException, or another existing exception class**. It's also recommended to provide both a no-argument constructor and a constructor with a message string, allowing for better error messaging and handling. For example:

```
public class FileFormatException extends
IOException {
   public FileFormatException() {}
   public FileFormatException(String message) {
            super(message);
       }
```

5.1.3. DECLARING CHECKED EXCEPTIONS

Any method that might result in a checked exception **must declare it in the method header using a throws clause**. For instance:

```
public void write(Object obj, String filename)
 throws IOException, ReflectiveOperationException
```

The throws clause lists the **exceptions that the method might throw**, either due to a throw statement within the method or because it calls another method that also has a throws clause.

Exceptions in the throws clause can be **combined into a common superclass**, but this should be done judiciously. For example, if a method can throw multiple subclasses of IOException, it's sensible to include them all under throws IOException. However, combining unrelated exceptions into throws Exception is discouraged as it defeats the purpose of exception checking.

When overriding a method, the **overriding method cannot throw more checked exceptions than those declared by the superclass method**. However, it can throw fewer exceptions.

You can document the exceptions thrown by a method using the **@throws tag** in Javadoc comments. This is particularly useful for documenting specific scenarios where exceptions might occur.

```
@throws NullPointerException if filename is
null
@throws FileNotFoundException if there is no
file with name filename
```

Note that **lambda expressions cannot specify the exception type**. If a lambda expression can throw a checked exception, it can only be passed to a functional interface method that declares that exception.

```
list.forEach(obj -> write(obj,"output.dat"));
// Error: write method might throw IOException
```

Here, the Consumer functional interface's accept method does not declare any checked exceptions, so passing a lambda expression that might throw an exception is not allowed.

5.1.4. CATCHING EXCEPTIONS

To catch an exception use a **try block** followed by one or more **catch blocks**. The basic structure is as follows:

```
try {
    // statements that may throw an exception
} catch (ExceptionClass ex) {
    // handler for the specific exception class
}
```

If an exception of the specified class occurs within the try block, **control is transferred to the corresponding catch block**. The **ex variable** refers to the exception object, which can be inspected by the handler if needed.

You can have multiple catch blocks to handle different exception classes:

```
try {
    // statements
} catch (ExceptionClass1 ex) {
    // handler for ExceptionClass1
} catch (ExceptionClass2 ex) {
    // handler for ExceptionClass2
} catch (ExceptionClass3 ex) {
    // handler for ExceptionClass3
}
```

The catch blocks are matched in order from top to bottom, so more specific exception classes should come first.

Alternatively, you can group multiple exception classes in a single catch block using the **multi-catch syntax**:

```
try {
    // statements
} catch (ExceptionClass1 | ExceptionClass2 |
  ExceptionClass3 ex) {
    // shared handler for multiple exception
       classes
}
```

In this case, the handler **can only invoke methods on the ex variable that are common to all exception classes caught**.

5.1.5. THE TRY-WITH-RESOURCES STATEMENT

Managing resources in exception handling scenarios can be tricky. Consider a situation where you're writing to a file and need to ensure it's properly closed even if an exception occurs. Without careful handling, resources may not be released, leading to potential issues such as lost output or resource exhaustion.

To address this problem, Java introduces the **try-with-resources statement**. This special form of the try statement **allows you to specify and automatically manage resources**. Here's the basic syntax:

```
try (ResourceType1 res1 = init1; ResourceType2
res2 = init2; ...) {
    // statements
}
```

Each resource must implement the **AutoCloseable interface**, which defines a **close() method**. When the try block exits, whether normally or due to an exception, the close() method of each resource is invoked in the reverse order of their initialization.

For example, consider writing to a file using a PrintWriter:

```
ArrayList<String> lines = ...;
try (PrintWriter out = new PrintWriter
("output.txt")){
    for (String line : lines) {
        out.println(line.toLowerCase());
    }
}
```

In this example, out.close() is guaranteed to be called, ensuring proper resource cleanup.

You can use **try-with-resources with multiple resources** as well:

```
try (Scanner in = new Scanner(Paths.get("/usr/
share/dict/words"));
     PrintWriter out = newPrintWriter
     ("output.txt")) {
    while (in.hasNext()) {
        out.println(in.next().toLowerCase());
    }
}
```

If a resource's close() method throws an exception, it will be suppressed if there's another exception already being thrown. The original exception takes precedence, and the suppressed exceptions can be retrieved using the getSuppressed() method.

```
try {
    // ...
} catch (IOException ex) {
    Throwable[] secondaryExceptions =
    ex.getSuppressed();
    // handle exceptions
}
```

Try-with-resources can also include catch clauses to handle any exceptions within the statement.

5.1.6. THE FINALLY CLAUSE

The **finally clause** is used to execute code that needs to be run regardless of whether an exception occurs or not within a try block. It's particularly useful for resource cleanup, like closing files or releasing locks.

```
try {
    // Do work
} finally {
    // Clean up
}
```

It's advised to avoid throwing exceptions within the finally block because if an exception occurs in both the try block and the finally block, the **one from the finally block will mask the original exception**.

Similarly, a finally block **should not contain a return statement**. If the try block has a return statement, the one in the finally block will replace the return value.

5.1.7. RETHROWING AND CHAINING EXCEPTIONS

When rethrowing exceptions, keep in mind the declaration of the enclosing method. The Java compiler tracks the flow of exceptions, so you typically don't need to change method signatures unless necessary.

Sometimes, you may want to change the class of a thrown exception, particularly to provide more meaningful information to higher-level code. You can catch the original exception and chain it to a new one. For example:

```
try {
    // Access the database
} catch (SQLException ex) {
    throw new ServletException("database
error", ex);
}
```

You can retrieve the original exception using getCause() method or initialize the cause using initCause() method if needed.

Chaining exceptions is also useful when you need to handle checked exceptions in methods that are not allowed to throw checked exceptions. You can catch the checked exception and chain it to an unchecked one.

5.1.8. THE STACK TRACE

When an exception is not caught anywhere in Java, a **stack trace** is displayed, which is essentially a **listing of all pending method calls at the point where the exception was thrown**. This stack trace is sent to **System.err**, the stream for error messages.

If you want to save the exception somewhere else, perhaps for inspection by your technical support staff, you can set the default uncaught exception handler using the **setDefaultUncaughtExceptionHandler** method of the Thread class. For example:

```
Thread.setDefaultUncaughtExceptionHandler((thre
ad, ex) -> {
    // Record the exception
});
```

Note that **an uncaught exception terminates the thread in which it occurred**. Therefore, if your application only has one thread, it will exit after invoking the uncaught exception handler.

Sometimes, you may be forced to catch an exception without knowing what to do with it. For instance, the **Class.forName** method throws a checked exception that you need to handle. In such cases, it's a good practice to at least print the stack trace:

```
try {
    Class<?> cl = Class.forName(className);
    // ...
} catch (ClassNotFoundException ex) {
    ex.printStackTrace();
}
```

If you want to store the stack trace of an exception, you can put it into a string. This can be achieved by using a **ByteArrayOutputStream** to capture the stack trace:

```
ByteArrayOutputStream out = new
                        ByteArrayOutputStream();
ex.printStackTrace(new PrintStream(out));
String description = out.toString();
```

Additionally, if you need to process the stack trace in more detail, you can call **getStackTrace()** on the exception object to obtain an array of **StackTraceElement** instances.

5.1.9. THE Objects.requireNonNull METHOD

The **Objects** class in Java provides a convenient method for null checks of parameters called **requireNonNull**. Here's a sample usage:

```
public void process(String directions) {
    this.directions =
            Objects.requireNonNull(directions);
    // ...
}
```

If directions is null, a NullPointerException is thrown. While this might not seem like a significant improvement at first, when working back from a stack trace, seeing a call to requireNonNull immediately indicates the source of the issue.

You can also supply a **message string for the exception**, which provides additional context:

```
this.directions =
          Objects.requireNonNull(directions,
          "directions must not be null");
```

This message will be included in the thrown NullPointerException, aiding in debugging and understanding the cause of the issue.

5.2. ASSERTIONS

Assertions are a commonly used idiom of **defensive programming**. They serve as a mechanism to validate internal assumptions during testing, ensuring that **certain conditions hold true before proceeding with execution**.

In Java, there are two forms of the assertion statement:

```
assert condition;
assert condition : expression;
```

The first form **evaluates the condition and throws an AssertionError if it is false**. The second form **allows for specifying an expression, which becomes the message of the error object if the condition is not met**.

For instance, to assert that a variable x is non-negative:

```
assert x >= 0;
```

Or, to pass the actual value of x into the AssertionError object for later display:

```
assert x >= 0 : x;
```

By default, assertions are disabled in Java. However, you can enable them by running the program with the **-ea** or **-enableassertions** option. For example:

```
java –ea MainClass
```

Enabling or disabling assertions does not require recompilation of the program, as it's handled by the class loader. When assertions are disabled, the class loader removes the assertion code to avoid slowing down execution.

You can enable assertions selectively for specific classes or packages using:

```
java –ea:MyClass –ea:com.mycompany.mylib...
MainClass
```

This command turns on assertions for MyClass and all classes in the com.mycompany.mylib package and its subpackages.

Conversely, you can disable assertions in certain classes or packages with:

```
java –ea:... –da:MyClass MainClass
```

Additionally, you can programmatically control assertion status using methods provided by the ClassLoader class.

Note

Assertions in Java are intended for debugging purposes to validate internal assumptions, not as a mechanism for enforcing contracts. For handling inappropriate parameters of public methods, it's recommended to throw an IllegalArgumentException instead of using assertions.

5.3. LOGGING

Logging in Java serves as a robust alternative to inserting print statements for debugging purposes. It allows programmers to gain insight into program behavior without cluttering the codebase with temporary debugging artifacts.

5.3.1. USING LOGGERS

The logging API in Java revolves around loggers. By default, there's a global logger accessible via **Logger.getGlobal()**. Logging an information message with this logger can be done using the info method:

```
Logger.getGlobal().info("Opening file " +
                                    filename);
```

This message includes automatic inclusion of time, calling class, and method names. However, logging can be globally turned off by setting the level of the global logger to **OFF**.

To minimize the cost of message creation when logging is disabled, lambda expressions can be used:

```
Logger.getGlobal().info(() -> "Opening file " +
                                    filename);
```

5.3.2. LOGGERS

In professional applications, it's common to define custom loggers. Loggers are created using **Logger.getLogger(name)**, and subsequent calls to the same name yield the same logger object. Logger names are hierarchical, similar to package names. Turning off messages to a parent logger also deactivates child loggers.

5.3.3. LOGGING LEVELS

There are seven logging levels in Java: **SEVERE**, **WARNING**, **INFO**, **CONFIG**, **FINE**, **FINER**, and **FINEST**.

By default, only the top three levels are logged. You can set a different threshold level using **setLevel(Level)** on a logger. Additionally, you can use **Level.ALL** to turn on logging for all levels or **Level.OFF** to turn off all logging.

Convenience methods exist for logging messages at different levels, such as warning and fine. If the level is variable, you can use the log method and supply the level dynamically.

5.3.4. OTHER LOGGING METHODS

Java's logging API provides convenience methods for tracing execution flow and logging unexpected exceptions.

Additionally, there are methods for logging exceptions along with their descriptions, such as **log** and **throwing**.

To ensure precise location information in log messages, the **logp** method can be used. Furthermore, if logging messages need to be localized for users in multiple languages, **logrb** methods can be utilized along with **ResourceBundle**.

5.3.5. LOGGING CONFIGURATION

In Java, various properties of the logging system can be changed by editing a configuration file.

The default configuration file is located at **jre/lib/logging.properties**.

However, you can specify another file by setting the **java.util.logging.config.file** property when starting your application:

```
java -Djava.util.logging.config.file=configFile
MainClass
```

Note

Setting System.setProperty("java.util.logging.config.file", configFile) in the main method has no effect because the log manager is initialized during VM startup, before main executes.

To change the default logging level, edit the configuration file and modify the line **.level=INFO**. Additionally, you can specify logging levels for your own loggers by adding lines like:

```
com.mycompany.myapp.level=FINE
```

Logger names are hierarchical, similar to package names, and settings apply to child loggers as well.

5.3.6. LOG HANDLERS

Loggers send log records to handlers, which determine where the logs are outputted. By default, loggers send records to a **ConsoleHandler**, which prints them to the **System.err** stream. You can also install your own handlers.

```
Logger logger =
       Logger.getLogger(“com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
handler.setLevel(Level.FINE);
logger.addHandler(handler);
```

This code creates a logger for a specific package, sets the logging level to FINE, and adds a console handler for it. Setting setUseParentHandlers(false) prevents logs from being sent to parent handlers.

5.3.7. FILTERS AND FORMATTERS

In Java logging, in addition to filtering by logging levels, each logger and handler can have an additional filter applied to them. This filter implements the **Filter interface**, which contains a single method:

```
boolean isLoggable(LogRecord record)
```

To install a filter into a logger or handler, you simply call the **setFilter method**. It's important to note that you can have **at most one filter at a time for each logger or handler**.

The **ConsoleHandler** and **FileHandler** classes in Java emit log records in text and XML formats by default. However, you have the flexibility to define your own log record formats. This is achieved by extending the **Formatter** class and overriding the **format** method:

```
String format(LogRecord record)
```

Within the **format** method, you have the freedom to format the log record in any desired way and return the resulting string. Additionally, you may want to use the **formatMessage** method within your formatter, which formats the message part of the record, substituting parameters and applying localization.

```
String formatMessage(LogRecord record)
```

Some file formats, such as XML, may require header and footer parts that surround the formatted records. To accommodate this, you can override the **getHead** and **getTail** methods:

```
String getHead(Handler h)
String getTail(Handler h)
```

Finally, to install your custom formatter into a handler, you call the **setFormatter** method.

GENERIC PROGRAMMING

You often need to implement classes and methods that work with multiple types. For example, an ArrayList<T> stores elements of an arbitrary class T. We say that the ArrayList class is **generic**, and T is a **type parameter**. The basic idea is very simple and incredibly useful.

6.1. GENERIC CLASSES

Generic classes in Java are **classes that can work with any data type**.

For instance, consider the Entry class, designed to store key/value pairs.

```
public class Entry<K, V> {
       private K key;
       private V value;
       public Entry(K key, V value) {
           this.key = key;
           this.value = value;
       }
       public K getKey() { return key; }
       public V getValue() { return value; }
   }
```

As you can see, the type parameters K and V are specified inside angle brackets after the name of the class.

You instantiate the generic class by substituting types for the type variables. For example,

```
Entry<String, Integer>
```

is an ordinary class with methods String getKey() and Integer getValue().

Caution

Type parameters cannot be instantiated with primitive types. For example, Entry<String, int> is not valid in Java.

When you construct an object of a generic class, you can omit the type parameters from the constructor. For example:

```
Entry<String, Integer> entry = new
Entry<>("Fred", 42);
```

Same as new Entry<String, Integer>("Fred", 42)

Note that you still provide an empty pair of angle brackets before the construction.

6.2. GENERIC METHODS

Generic methods are **methods that can operate on any data type**. They can be methods of a regular class or a generic class. Here is an example of a generic method in a class that is not generic:

```
public class Arrays {
  public static <T> void swap(T[] array, int i,
  int j) {
      T temp = array[i];
      array[i] = array[j];
      array[j] = temp;
} }
```

This swap method can be used to swap elements in an arbitrary array, as long as the array element type is not a primitive type.

```
   String[] friends = ...;
   Arrays.swap(friends, 0, 1);
```

When you declare a generic method, the type parameter is placed after the modifiers (such as public and static) and before the return type:

```
public static <T> void swap(T[] array, int i,
int j)
```

When calling a generic method, you do not need to specify the type parameter. It is inferred from the method parameter and return types. For example, in the call Arrays.swap(friends, 0, 1), the type of friends is String[], and the compiler can infer that T should be String.

You can, if you like, supply the type explicitly, before the method name, like this:

```
Arrays.<String>swap(friends, 0, 1);
```

6.3. TYPE BOUNDS

Type bounds in Java generics allow you to **impose restrictions on the type parameters of a generic class or method**. These bounds ensure that the type parameters meet certain requirements, such as extending specific classes or implementing particular interfaces.

For instance, consider a scenario where you have an ArrayList containing objects of a class that implements the AutoCloseable interface, and you want to close all elements in the list. You can define a method like this:

```
public static <T extends AutoCloseable> void
closeAll(ArrayList<T> elems) throws Exception {
    for (T elem : elems)
        elem.close();
}
```

Here, the type parameter T is bounded by AutoCloseable, **ensuring that elements in the ArrayList are subtypes of AutoCloseable**, allowing the invocation of the close() method on each element.

You can specify multiple bounds for a type parameter, using the syntax T extends Runnable & AutoCloseable. In this case, T must be a subtype of both Runnable and AutoCloseable. However, if a class is among the bounds, it must be the first one listed.

Type bounds are essential when dealing with collections like ArrayList because, unlike arrays, they do not support covariance. For example, ArrayList<Manager> is not a subtype of ArrayList<Employee>. To handle this, you can use bounded type parameters. Otherwise, you could risk corrupting the list by adding incompatible elements.

6.4. TYPE VARIANCE AND WILDCARDS

Wildcards are used to specify how method parameter and return types should vary. This approach, known as **use-site variance**, helps ensure type safety. Unlike arrays, which are covariant, **generic types in Java are invariant**. Wildcards provide a mechanism for flexible type handling, preventing potential type-related errors.

There are two main types of wildcard usage: subtype wildcards and supertype wildcards.

6.4.1. SUBTYPE WILDCARDS

A subtype wildcard, denoted by **? extends Employee**, represents an unknown subtype of Employee. This is useful when you want to **process elements of a collection without modifying it**. For example, consider the method printNames:

```
public static void printNames(ArrayList<?
extends Employee> staff) {
    for (int i = 0; i < staff.size(); i++) {
        Employee e = staff.get(i);
        System.out.println(e.getName());
    }
}
```

Here, **? extends Employee** allows you to pass an ArrayList<Employee> or any subtype of Employee to the method. You can read elements from such a list, but you cannot write to it. Attempting to add an element to ArrayList<? extends Employee> will result in a **compilation error because the compiler cannot guarantee the specific subtype**.

6.4.2. SUPERTYPE WILDCARDS

A supertype wildcard, denoted by **? super Employee**, represents an arbitrary supertype of Employee. This is useful when you want to allow more flexibility in accepting parameters, especially for functional objects. For example, consider the method printAll:

```
public static void printAll(Employee[] staff,
Predicate<? super Employee> filter) {
    for (Employee e : staff)
        if (filter.test(e))
            System.out.println(e.getName());
}
```

Here, **? super Employee** allows you to pass a Predicate<Employee> or any supertype of Employee to the method. This is particularly handy when dealing with functional interfaces where parameter types are naturally contravariant.

Use extends when you're consuming values from a collection and super when you're producing values into it.

6.4.3. WILDCARDS WITH TYPE VARIABLES

Consider a generalization of the method of the preceding section that prints arbitrary elements fulfilling a condition:

```
public static <T> void printAll(T[] elements,
Predicate<T> filter) {
      for (T e : elements)
          if (filter.test(e))
              System.out.println(e.toString());
}
```

This is a generic method that works for arrays of any type. The type parameter is the type of the array that is being passed. However, it suffers from the limitation that you saw in the preceding section. **The type parameter of Predicate must exactly match the type parameter of the method**.

The solution is the same — but this time, the bound of the wildcard is a type variable:

```
public static <T> void printAll(T[] elements,
Predicate<? super T> filter)
```

This method takes a filter for elements of type T or any supertype of T.

6.4.4. UNBOUNDED WILDCARDS

It is possible to have unbounded wildcards for situations where **you only do very generic operations**. For example, here is a method to check whether an ArrayList has any null elements:

```
public static boolean hasNulls(ArrayList<?>
elements) {
       for (Object e : elements) {
           if (e == null) return true;
       }
       return false;
   }
```

Since the type parameter of the ArrayList doesn't matter, it makes sense to use an ArrayList<?>.

6.4.5 WILDCARD CAPTURE

Let's try to define a swap method using wildcards:

```
public static void swap(ArrayList<?> elements,
int i, int j) {
   ? temp = elements.get(i);    // Won't work
   elements.set(i, elements.get(j));
   elements.set(j, temp);
}
```

That won't work. You can use ? as a type argument, but not as a type.

However, there is a workaround. Add a helper method, like this:

```
public static void swap(ArrayList<?> elements,
int i, int j) {
    swapHelper(elements, i, j);
}
private static <T> void swapHelper(ArrayList<T>
elements, int i, int j) {
    T temp = elements.get(i);
    elements.set(i, elements.get(j));
```

```
    elements.set(j, temp);
  }
```

The call to swapHelper is valid because of a special rule called **wildcard capture**. The compiler doesn't know what ? is, but it stands for some type, so it is OK to call a generic method. The type parameter T of swapHelper "captures" the wildcard type. Since swapHelper is a generic method, not a method with wildcards in parameters, it can make use of the type variable T to declare variables.

6.5 GENERICS IN THE JAVA VIRTUAL MACHINE

When generic types and methods were added to Java, the Java designers wanted the generic forms of classes to be compatible with their preexisting versions. For example, it should be possible to pass an ArrayList<String> to a method from pregeneric days that accepted the ArrayList class, which collects elements of type Object. The language designers decided on an implementation that "erases" the types in the virtual machine.

6.5.1 TYPE ERASURE

When you define a generic type, it is compiled into a **raw type**. For example, the Entry<K, V> class turns into:

```
public class Entry {
   private Object key;
   private Object value;
   public Entry(Object key, Object value {
         this.key = key;
         this.value = value;
   }
   public Object getKey() { return key; }
   public Object getValue() { return value; }
}
```

Every K and V is replaced by **Object**.

If a type variable has bounds, it is **replaced with the first bound**. Suppose we declare the Entry class as

```
public class Entry<K extends Comparable<? super
K> & Serializable, V extends Serializable>
```

Then it is erased to a class

```
public class Entry {
   private Comparable key;
   private Serializable value; ...
}
```

6.5.2 CAST INSERTION

Suppose your program compiled with "unchecked" warnings, perhaps because you used casts or mixed generic and raw Entry types. Then it is possible for an Entry<String, Integer> to have a key of a different type.

Therefore, it is also necessary to have safety checks at runtime. The compiler inserts a **cast whenever one reads from an expression with erased type**. Consider, for example:

```
   Entry<String, Integer> entry = ...;
   String key = entry.getKey();
```

Since the erased getKey method returns an Object, the compiler generates code equivalent to

```
   String key = (String) entry.getKey();
```

6.6 RESTRICTIONS ON GENERICS

There are several restrictions when using generic types and methods in Java. Most of them are consequences of type erasure.

6.6.1 NO PRIMITIVE TYPE ARGUMENTS

A type parameter can never be a primitive type. For example, you cannot form an ArrayList<int>. As you have seen, in the virtual machine there is only one type, the raw ArrayList that stores elements of type Object. An int is not an object.

6.6.2 AT RUNTIME, ALL TYPES ARE RAW

In the virtual machine, there are only raw types. For example, you cannot inquire at runtime whether an ArrayList contains String objects. A condition such as

```
   if (a instanceof ArrayList<String>)
```

is a compile-time error since no such check could ever be executed.

A cast to an instantiation of a generic type is equally ineffective, but it is legal.

```
Object result = ...;
ArrayList<String> list = (ArrayList<String>)
                         result;
```

Warning: this only checks whether result is a raw ArrayList

6.6.3 YOU CANNOT INSTANTIATE TYPE VARIABLES

You cannot use type variables in expressions such as new T(...) or new T[...]. These forms are outlawed because they would not do what the programmer intends when T is erased.

If you want to create a generic instance or array, you'd like the element type of the array to be the same as the type of obj. This attempt does not work:

```
public static <T> T[] repeat(int n, T obj) {
   T[] result = new T[n];
   // Error—cannot construct an array new T[...]
   for (int i = 0; i < n; i++)
          result[i] = obj;
   return result;
}
```

To solve this problem, ask the caller to provide the array constructor as a method reference:

```
String[] greetings = Arrays.repeat(10, "Hi",
String[]::new);
```

Here is the implementation of the method:

```
public static <T> T[] repeat(int n, T obj,
IntFunction<T[]> constr) {
   T[] result = constr.apply(n);
   for (int i = 0; i < n; i++) result[i] = obj;
       return result;
}
```

Alternatively, you can ask the user to supply a class object, and use reflection.

```
public static <T> T[] repeat(int n, T obj,
Class<T> cl) {
    @SuppressWarnings("unchecked") T[] result
      =(T[])java.lang.reflect.Array.newInstanc
                                    e(cl, n);
    for(int i = 0; i < n; i++) result[i] = obj;
       return result;
}
```

This method is called as follows:

```
String[] greetings = Arrays.repeat(10, "Hi",
String.class);
```

Another option is to ask the caller to allocate the array. Usually, the caller is allowed to supply an array of any length, even zero. If the supplied array is too short, the method makes a new one, using reflection.

```
public static <T> T[] repeat(int n, T obj, T[]
array) {
   T[] result;
   if (array.length >= n)
     result = array;
   else {
     @SuppressWarnings("unchecked") T[]newArray
     =(T[])java.lang.reflect.Array.newInstance(
       array.getClass().getComponentType(), n);
     result = newArray;
   }
   for (int i = 0; i < n; i++) result[i] = obj;
     return result;
 }
```

6.6.4 YOU CANNOT CONSTRUCT ARRAYS OF PARAMETERIZED TYPES

Suppose you want to create an array of Entry objects:

```
Entry<String, Integer>[] entries = new
Entry<String, Integer>[100];
```

cannot construct an array with generic component type

This is a syntax error. The construction is outlawed because, after erasure, the array constructor would create a raw Entry array. It would then be possible to add Entry objects of any type (such as Entry<Employee, Manager>) without an ArrayStoreException.

6.6.5 CLASS TYPE VARIABLES ARE NOT VALID IN STATIC CONTEXTS

Consider a generic class with type variables, such as Entry<K, V>. **You cannot use the type variables K and V with static variables or methods**. For example, the following does not work:

```
public class Entry<K, V> {
   private static V defaultValue;
```

V in static context

```
   public static void setDefault(V value)
                       { defaultValue = value; }
```

V in static context ...

```
}
```

After all, type erasure means there is only one such variable or method in the erased Entry class, and not one for each K and V.

6.6.6 METHODS MAY NOT CLASH AFTER ERASURE

You may not declare methods that would cause clashes after erasure. For example, the following would be an error:

```
public interface Ordered<T> extends
Comparable<T> {
    public default boolean equals(T value) {
    // Error—erasureclasheswith Object.equals
       return compareTo(value) == 0;
    }
    ...
}
```

The equals(T value) method erases to equals(Object value), which clashes with the same method from Object.

6.6.7 EXCEPTIONS AND GENERICS

You cannot throw or catch objects of a generic class. In fact, you cannot even form a generic subclass of Throwable:

```
public class Problem<T> extends Exception
```

Error: a generic class can't be a subtype of Throwable

You cannot use a type variable in a catch clause:

```
public static <T extends Throwable> void
doWork(Runnable r, Class<T> cl) {
       try {r.run();
} catch (T ex) {
       // Error—can't catch type variable
       Logger.getGlobal().log(..., ..., ex);
       }
}
```

However, you can have a type variable in the throws declaration:

```
public static <V, T> V doWork(Callable<V> c, T
ex) throws T {
       try {
           return c.call();
       } catch (Throwable realEx) {
           ex.initCause(realEx);
throw ex; }
}
```

6.7 REFLECTION AND GENERICS

6.7.1 THE CLASS<T> CLASS

The **Class** class has a type parameter, namely the class that the Class object describes.

Consider the String class. In the virtual machine, there is a Class object for this class, which you can obtain as "Fred".getClass() or, more directly, as the class literal String.class. You can use that object to find out what methods the class has, or to construct an instance.

String.class has type Class<String>: Its newInstance method returns a String.

That information can **save you a cast**. Consider this method:

```
public static <T> ArrayList<T> repeat(int n,
Class<T> cl)throws ReflectiveOperationException
{
   ArrayList<T> result = new ArrayList<>();
   for (int i = 0; i < n; I++)
      result.add(cl.newInstance());
   return result;
}
```

The method compiles since cl.newInstance() returns a result of type T.

6.7.2 GENERIC TYPE INFORMATION IN THE VIRTUAL MACHINE

Erasure only affects instantiated type parameters. Complete information about the declaration of generic classes and methods is available at runtime.

The interface Type in the java.lang.reflect package represents generic type declarations. The interface has the following subtypes:

- The **Class** class, describing concrete types
- The **TypeVariable** interface, describing type variables (such as T extends Comparable<? super T>)
- The **WildcardType** interface, describing wildcards (such as ? super T)
- The **ParameterizedType** interface, describing generic class or interface types (such as Comparable<? super T>)
- The **GenericArrayType** interface, describing generic arrays (such as T[])

Both classes and methods can have type variables. Technically speaking, constructors are not methods, and they are represented by a separate class in the reflection library. They too can be generic. To find out whether a Class, Method, or Constructor object comes from a generic declaration, call the **getTypeParameters** method. You get an array of TypeVariable instances, one for each type variable in the declaration, or an array of length 0 if the declaration was not generic.

The TypeVariable<D> interface is generic. The type parameter is Class<T>, Method, or Constructor<T>, depending on where the type variable was declared.

COLLECTIONS

7.1 AN OVERVIEW OF THE COLLECTIONS FRAMEWORK

The Java Collections Framework provides **implementations of common data structures**, offering common interfaces for coding independence.

The primary interface is **Collection**, which is generic with a type parameter E for element type. Other interfaces include **List<E>**, **Map<K, V>**.

It's **recommended to use interfaces in code for flexibility**. For instance, instead of directly referencing an ArrayList, store its reference in a variable of type List:

```
List<String> words = new ArrayList<>();
```

The framework eliminates the need to reinvent basic algorithms. The Collection interface offers basic algorithms like **addAll** and **removeIf**, while the Collections utility class provides additional algorithms for **sorting**, **shuffling**, etc.

7.2 ITERATORS

Each collection offers a method to iterate through its elements.

The **Iterable<T>** interface defines the iterator() method to yield an iterator that you can use to visit all elements.

```
Collection<String> coll = ...;
Iterator<String> iter = coll.iterator();
while (iter.hasNext()) {
    String element = iter.next();
    // Process element
}
```

Alternatively, the enhanced for loop simplifies iteration:

```
for (String element : coll) {
    // Process element
}
```

Note

For any object c of a class that implements the Iterable<E> interface, the enhanced for loop is translated to the preceding form.

The Iterator interface also includes a **remove()** method to remove the previously visited element. Additionally, the **removeIf** method can be used to remove elements based on a condition. However, care must be taken with the remove() method as **it removes the last element returned by the iterator, not the element to which the iterator currently points**.

The **ListIterator** interface, a subinterface of Iterator, is useful for working with linked lists. It includes methods for adding an element before the iterator, setting the visited element to a different value, and navigating backward. However, caution must be exercised when using multiple iterators on a data structure, as mutation by one iterator can invalidate others, potentially leading to ConcurrentModificationException.

7.3 SETS

Sets offer efficient membership testing but do not maintain the order of elements. They are particularly useful when the order of elements is irrelevant, such as in disallowing certain words as usernames.

The Set interface is implemented by **HashSet** and **TreeSet** classes, which internally employ different mechanisms.

Generally, **HashSet**s are a bit more efficient, provided you have a good hash function for your elements. Library classes such as String or Path have good hash functions. For example, that set of bad words can be implemented simply as:

```
Set<String> badWords = new HashSet<>();
badWords.add(“sex”);
badWords.add(“drugs”);
badWords.add(“c++”);
if (badWords.contains(username.toLowerCase()))
    System.out.println(“Please choose a
    different user name”);
```

You use a **TreeSet** if you want to traverse the set in sorted order. One reason you might want to do this is to present users a sorted list of choices.

The element type of the set **must implement the Comparable interface**, or you need to supply a Comparator in the constructor.

```
TreeSet<String> countries = new TreeSet<>();
// Visits added countries in sorted order
countries = new TreeSet<>((u, v) ->
u.equals(v) ? 0
: u.equals(“USA”) ? -1
: v.equals(“USA”) ? 1
: u.compareTo(v));
// USA always comes first
```

7.4 MAPS

Maps store associations between keys and values.

Call **put** to add a new association, or change the value of an existing key:

```
Map<String,
Integer> counts = new HashMap<>);
counts. put("Alice", 1);
// Adds the key/value pair to the map counts
put ("Alice", 2);
// Updates the value for the key
```

This example uses a **hash map** which, as for sets, is usually the better choice if you don't need to visit the keys in sorted order. If you do, use a **TreeMap** instead.

Here is how you can get the value associated with a key:

```
int count = counts get ("Alice");
```

If the key isn't present, the get method returns null. In this example, that would cause a NullPointer Exception when the value is unboxed. A better alternative is:

```
int count = counts. getOrDefault ("Alice", 0);
```

Then a count of 0 is returned if the key isn't present.

When you update a counter in a map, you first need to check whether the counter is present, and if so, add 1 to the existing value. The **merge** method simplifies that common operation. The call:

```
counts. merge(word, 1, Integer:: sum);
```

associates word with 1 if the key wasn't previously present, and otherwise combines the previous value and 1, using the Integer:: sum function.

You can **get views of the keys, values, and entries of a map** by calling these methods:

```
Set<k> keySet ( )
Set<Map. Entry<K, V>> entrySet ( )
Collection<K> values ()
```

The collections that are returned are not copies of the map data, but they are **connected to the map**. If you remove a key or entry from the view, then the entry is also removed from the underlying map.

To **iterate through all keys and values of a map**, you can iterate over the set returned by the entrySet method:

```
for (Map. Entry<String, Integer> entry: counts. entrySet ()) {
   String k = entry getKey();
   Integer v = entry getValue();
   Process k, v
}
```

Or simply use the forEach method:

```
counts. forEach((k, v) -> {
   Process k, v
});
```

Caution

Some map implementations (for example, ConcurrentHashMap) **disallow null for keys or values**. And with those that allow it (such as HashMap), you need to be very careful if you do use null values. A number of map methods interpret a null value as an indication that an entry is absent, or should be removed.

7.5 OTHER COLLECTIONS

In this section, several collection classes that offer practical utility are discussed briefly.

7.5.1 PROPERTIES

The **Properties class** implements a map that can be easily saved and loaded using a plain text format, commonly used for storing program configuration options.

```
Properties settings = new Properties();
settings.put("width", "200");
settings.put("title", "Hello, World!");
try (OutputStream out = Files.newOutputStream(path)) {
    settings.store(out, "Program Properties");
}
```

Property files are encoded in ASCII, with comments starting with # or !. Certain characters are escaped, and a newline in a key or value is represented as \n.

To **load properties from a file**:

```
try (InputStream in = Files.newInputStream(path)) {
    settings.load(in);
}
```

Then, **getProperty** method can be used to r**etrieve a value for a key**, with an option to specify a default value:

```
String title = settings.getProperty("title", "New Document");
```

7.5.2 BIT SETS

The **BitSet class** stores a sequence of bits efficiently by packing them into an array of long values. It's useful for sequences of flag bits or representing sets of non-negative integers.

```
BitSet bitSet = new BitSet();
bitSet.set(1);
bitSet.set(3);
```

7.5.3 ENUMERATION SETS AND MAPS

For collecting sets of enumerated values, **EnumSet** is preferred over BitSet. EnumSet has no public constructors; static factory methods are used for construction.

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY,
               THURSDAY, FRIDAY, SATURDAY,
                SUNDAY }
Set<Weekday> always = EnumSet.allOf(Weekday.class);
Set<Weekday> never = EnumSet.noneOf(Weekday.class);
Set<Weekday> workday = EnumSet.range(Weekday.MONDAY, Weekday.FRIDAY);
Set<Weekday> mwf = EnumSet.of(Weekday.MONDAY, Weekday.WEDNESDAY, Weekday.FRIDAY);
```

EnumMap is a map with keys belonging to an enumerated type, implemented as an array of values.

```
EnumMap<Weekday, String> personInCharge = new EnumMap<>(Weekday.class);
personInCharge.put(Weekday.MONDAY, "Fred");
```

These classes allow convenient manipulation of sets and maps with enumerated keys or values.

7.5.4 STACKS, QUEUES, DEQUES, AND PRIORITY QUEUES

Stack: A data structure for adding and removing elements at one end (the "top" of the stack). Adding elements in the middle is not supported. Java has a legacy Stack class, but it's recommended to use **ArrayDeque** instead.

```
ArrayDeque<String> stack = new ArrayDeque<>();
```

```
stack.push("Peter");
stack.push("Paul");
stack.push("Mary");

while (!stack.isEmpty())
    System.out.println(stack.pop());
```

Queue: Allows efficient addition of elements at one end (the "tail") and removal from the other end (the "head"). Use ArrayDeque for a non-thread-safe queue.

```
Queue<String> queue = new ArrayDeque<>();
queue.add("Peter");
queue.add("Paul");
queue.add("Mary");

while (!queue.isEmpty())
    System.out.println(queue.remove());
```

Priority Queue: Retrieves elements in sorted order after they were inserted. A typical use case is job scheduling, where each job has a priority.

```
public class Job implements Comparable<Job>
{ ... }

PriorityQueue<Job> jobs = newPriorityQueue<>();
jobs.add(new Job(4, "Collect garbage"));
jobs.add(new Job(9, "Match braces"));
jobs.add(new Job(1, "Fix memory leak"));

while (jobs.size() > 0) {
    Job job = jobs.remove();
    // The most urgent jobs are removed first
    execute(job);
}
```

Priority queues can hold elements of a class that implements the Comparable interface or accept a Comparator in the constructor. Unlike TreeSet, iterating over elements does not guarantee a sorted order; the priority queue maintains its order internally without sorting all elements.

7.5.5 WEAK HASH MAPS

The **WeakHashMap** class addresses the issue of managing values whose keys are no longer in use within a program.

Problem Overview: When a key is no longer referenced in the program, the associated value should ideally be removed by the garbage collector. However, as long as the map object containing the key-value pairs is live, all entries within it are also considered live, preventing their reclamation by the garbage collector.

Solution: WeakHashMap resolves this problem by cooperating with the garbage collector to remove key-value pairs when the only remaining reference to a key is within the hash table entry.

Technical Implementation: WeakHashMap uses weak references to hold keys.

A WeakReference object holds a reference to another object, in this case, a hash table key.

Objects held by weak references are treated specially by the garbage collector. If an object is only reachable through a weak reference, the garbage collector reclaims the object and places the weak reference into a queue associated with the WeakReference object.

When a method is invoked on the WeakHashMap, it checks its queue of weak references for new arrivals and removes the associated entries, allowing the garbage collector to clean up the unreferenced key-value pairs.

7.6 VIEWS

A **collection** view is a lightweight object that implements a collection interface but **doesn't store elements**.

Examples include the **keySet** and values methods of a map, which provide views into the map. Another instance is the **Arrays.asList** method, which returns a List backed by an array's elements.

Usually, a view does not support all operations of its interface. For instance, calling add on a key set of a map or the list returned by Arrays.asList doesn't make sense.

7.6.1 RANGES

You can create a **sublist view of a list using the subList method**. For example:

```
List<String> sentence = ...;
List<String> nextFive = sentence.subList(5, 10);
```

This view accesses elements with indices 5 through 9, and any mutations to the sublist affect the original list.

For sorted sets and maps, you specify a range by the lower and upper bounds using methods like subSet. The range's first bound is inclusive, while the second is exclusive.

7.6.2 EMPTY AND SINGLETON VIEWS

The Collections class provides static methods yielding **immutable empty and singleton views for various collections**. For instance:

```
doWork(Collections.emptyMap());
doWork(Collections.singletonMap("id", id));
```

These methods avoid creating heavyweight objects like HashMap or TreeMap when you need an empty or singleton collection.

7.6.3 UNMODIFIABLE VIEWS

An unmodifiable view allows you to **share the contents of a collection without allowing modifications**. For example:

```
public class Person {
  private ArrayList<Person> friends;
  public List<Person> getFriends() {
   return Collections.unmodifiableList(friends);
```

```
    }
}
```

Mutator methods throw exceptions when invoked on an unmodifiable view.

Note on Checked Views

To debug problems related to "heap pollution," where incorrect elements are inserted into a generic collection, you can use a checked view:

```
List<String> strings =
Collections.checkedList(new ArrayList<>(),
String.class);
```

This view monitors insertions into the list and throws an exception if an object of the wrong type is added.

Note on Synchronized Views

The Collections class produces synchronized views for safe concurrent access to data structures. However, classes from java.util.concurrent are recommended for concurrent access instead of synchronized views.

STREAMS

Streams provide a view of data that lets you specify computations at a higher conceptual level than with collections. With a stream, you specify what you want to have done, not how to do it. You leave the scheduling of operations to the implementation.

8.1 FROM ITERATING TO STREAM OPERATIONS

Traditionally, when processing a collection, you'd iterate over its elements and perform some action on each. Let's illustrate this with an example: counting long words in a book. Initially, we gather the words into a list:

```
String contents = new
String(Files.readAllBytes(Paths.get("alice.txt"
)), StandardCharsets.UTF_8);
// Read file into  string
List<String> words =
Arrays.asList(contents.split("\\PL+"));
// Split into words; non-letters are delimiters
```

Then, we iterate over the list to count the long words:

```
int count = 0;
for (String w : words) {
    if (w.length() > 12) count++;
}
```

With Java Streams, this operation can be expressed as:

```
long count = words.stream()
    .filter(w -> w.length() > 12)
    .count();
```

Notice how the stream code clearly indicates its purpose without the need for meticulous scanning. The method names convey the intentions directly. Additionally, while a loop dictates the order of operations explicitly, **a stream can optimize the sequence of operations as long as it delivers the correct outcome**.

By simply converting stream() to parallelStream(), the stream library can execute filtering and counting operations concurrently:

```
long count = words.parallelStream()
    .filter(w -> w.length() > 12)
    .count();
```

Streams adhere to the **"what, not how"** principle. In our stream example, we specify what needs to be done: identify long words and count them. We leave the details of execution, such as order and threading, to the stream implementation. Conversely, the loop at the start of this section dictates precisely how the computation should proceed, limiting optimization opportunities.

Though streams share similarities with collections, significant **distinctions** exist:

- **Storage:** Streams don't store elements themselves; they may originate from a collection or generate elements as needed.

- **Immutability:** Stream operations don't alter the source; instead, they produce new streams with transformations.
- **Laziness:** Stream operations delay execution until necessary, allowing for optimizations. For instance, if only the first five long words are required, the filter operation halts after finding them. This enables the possibility of infinite streams.

Returning to our example, stream() or parallelStream() creates a stream from the words list. filter() then produces another stream with long words, and count() acts as the terminal operation, yielding the result.

Working with streams typically involves a three-stage pipeline:

- **Creation:** Initiate a stream.
- **Transformation:** Specify intermediate operations to modify the stream.
- **Termination:** Apply a terminal operation to produce a final result, rendering the stream unusable thereafter.

In our example, stream() or parallelStream() initializes the stream, filter() transforms it, and count() serves as the terminal operation.

8.2 CREATING STREAMS

In Java, you can easily **convert collections into streams** using the **stream() method from the Collection interface**. Alternatively, if you have an array, utilize the static Stream.of() method.

```
Stream<String> words =
Stream.of(contents.split("\\PL+"));
// split returns a String[] array
```

The **of()** method accepts a variable number of arguments, enabling you to **construct a stream from any number of elements**.

```
Stream<String> song = Stream.of("gently",
                        "down", "the", “stream");
```

For **creating a stream from a portion of an array**, employ **Arrays.stream(array, from, to)**.

To **create a stream with no elements**, employ the static method **Stream.empty()**.

```
Stream<String> silence = Stream.empty();
```

The Stream interface offers two static methods to **generate infinite streams**. The **generate()** method takes a function with no arguments (or a Supplier<T> object) and produces a value whenever a stream value is needed.

```
Stream<String> echos = Stream.generate(() ->
"Echo"); // Stream of constant values
```

Similarly, for **generating a stream of random numbers**:

```
Stream<Double> randoms =
Stream.generate(Math::random);
// Stream of random numbers
```

For **creating infinite sequences**, such as 0, 1, 2, 3..., use the **iterate()** method. It requires a "seed" value and a function to apply repeatedly to the previous result.

```
Stream<BigInteger> integers =
Stream.iterate(BigInteger.ZERO, n ->
n.add(BigInteger.ONE));
```

In this sequence, the seed value is BigInteger.ZERO, followed by f(seed), which is 1 (as a big integer), then f(f(seed)), which is 2, and so forth.

8.3 FILTERING, MAPPING, AND FLATTENING STREAMS

Stream transformations allow you to **derive a new stream from an existing one**. We've previously encountered the **filter transformation**, which produces a stream containing elements that match a specific condition. For instance, to obtain a stream of only long words:

```
List<String> words = ...;
Stream<String> longWords =
words.stream().filter(w -> w.length() > 12);
```

The filter method takes a Predicate<T> as its argument, which is a function from type T to boolean.

Often, you'll need to **alter the values in a stream**. This is where the **map method** comes into play. It applies a function to each element and returns a new stream with the transformed values. For instance, to convert all words to lowercase:

```
Stream<String> lowercaseWords =
words.stream().map(String::toLowerCase);
```

Alternatively, you can use a **lambda expression**:

```
Stream<String> firstLetters =
words.stream().map(s -> s.substring(0, 1));
```

Here, the resulting stream contains the first letter of each word.

Now, suppose you have a function that doesn't return a single value but a stream of values, such as:

```
public static Stream<String> letters(String s){
    List<String> result = new ArrayList<>();
    for (int i = 0; i < s.length(); i++)
        result.add(s.substring(i, i + 1));
    return result.stream();
}
```

For example, letters("boat") returns the stream ["b", "o", "a", "t"].

If you apply the letters method to a stream of strings using map, you'll end up with a stream of streams:

```
Stream<Stream<String>> result =
words.stream().map(w -> letters(w));
```

To **flatten** this nested stream structure into a single stream of letters, use the **flatMap** method instead of map:

```
Stream<String> flatResult =
words.stream().flatMap(w -> letters(w));
```

Calls letters on each word and flattens the results

8.4 EXTRACTING SUBSTREAMS AND COMBINING STREAMS

You can **manipulate streams to extract substreams or combine multiple streams** in various ways.

The **limit(n)** method returns a new stream containing the first n elements (or all elements if the original stream is shorter). This is particularly useful for truncating infinite streams. For example:

```
Stream<Double> randoms =
Stream.generate(Math::random).limit(100);
```

Generates a stream with 100 random numbers

Conversely, the **skip(n)** method discards the first n elements of a stream. This is handy, for instance, when dealing with the result of splitting a string, where the first element is often an unwanted empty string:

```
Stream<String> words =
Stream.of(contents.split("\\PL+")).skip(1);
```

Skips the first empty string

To concatenate two streams, you can use the **concat** method from the Stream class:

```
Stream<String> combined = Stream.concat(
letters("Hello"), letters("World"));
```

Yields the stream ["H", "e", "l", "l", "o", "W", "o", "r", "l", "d"]

8.5 OTHER STREAM TRANSFORMATIONS

The **distinct** method returns a stream that preserves the order of elements from the original stream but eliminates duplicates:

```
Stream<String> uniqueWords =
Stream.of("merrily", "merrily", "merrily",
"gently").distinct();
```

Only one "merrily" is retained

For sorting a stream, there are variations of the **sorted** method. Here's an example where strings are sorted so that the longest ones come first:

```
Stream<String> longestFirst =
words.stream().sorted(Comparator.comparing(Stri
ng::length).reversed());
```

As with all stream transformations, the sorted method returns a new stream with elements sorted accordingly.

The **peek** method provides a way to execute a function for each element in the stream, mainly for debugging purposes:

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
  .peek(e -> System.out.println("Fetching " + e))
  .limit(20).toArray();
```

This allows you to observe the processing of an infinite stream in a lazy manner, facilitating debugging. You can even set breakpoints within the method passed to peek for more detailed debugging.

8.6 SIMPLE REDUCTIONS

Simple reductions are **terminal operations that reduce the stream to a non-stream value usable in your program**.

You're already familiar with a basic reduction: the **count** method, which returns the number of elements in a stream. Other simple reductions include **max** and **min**, which return the largest or smallest value, respectively.

However, there's a slight twist—**these methods return an Optional<T> value**. This wrapper either contains the answer or indicates that there is none (if the stream happens to be empty). Unlike the older approach of returning null, which could lead to null pointer exceptions in incompletely tested programs,

Here's how you can obtain the maximum value from a stream:

```
Optional<String> largest =
words.max(String::compareToIgnoreCase);
System.out.println("largest: " +
largest.orElse(""));
```

The **findFirst** method returns the first value in a non-empty collection. It's often useful when combined with filter. For instance, to find the first word starting with the letter Q:

```
Optional<String> startsWithQ = words.filter(s
-> s.startsWith("Q")).findFirst();
```

If any match suffices, not just the first one, you can use the **findAny** method. This is particularly effective when parallelizing the stream, as it can report any match found instead of being restricted to the first one:

```
Optional<String> startsWithQ =
words.parallel().filter(s ->
s.startsWith("Q")).findAny();
```

For determining if there's any match, you can utilize **anyMatch**. This method accepts a predicate argument, eliminating the need for a separate filter operation:

```
boolean aWordStartsWithQ =
words.parallel().anyMatch(s ->
s.startsWith("Q"));
```

Additionally, there are **allMatch** and **noneMatch** methods, which return true if all or no elements match a predicate, respectively. These methods also benefit from parallel execution.

8.7 THE OPTIONAL TYPE

8.7.1 HOW TO WORK WITH OPTIONAL VALUES

Effectively utilizing Optional involves employing methods that **either produce an alternative value if the original value is absent** or **consume the value only if it's present**.

Let's explore the first strategy. Often, you might have a default value that you wish to use when there's no match, such as the empty string:

```
String result = optionalString.orElse("");
```

Alternatively, you can invoke code to compute the default:

```
String result = optionalString.orElseGet(() -> System.getProperty("user.dir"));
// The function is only called when needed
```

Another approach is to throw an exception if there's no value:

```
String result = optionalString.orElseThrow(IllegalStateException::new);
// Supply a method that yields an exception object
```

The second strategy for working with optional values involves consuming the value only if it's present.
The **ifPresent** method accepts a function. If the optional value exists, it's passed to that function. Otherwise, nothing happens.

```
optionalValue.ifPresent(v -> process(v));
```

For instance, to add the value to a set if it's present:

```
optionalValue.ifPresent(v -> results.add(v));
// Or simply
optionalValue.ifPresent(results::add);
```

When calling ifPresent, no value is returned from the function. If you need to process the function result, **use map instead**:

```
Optional<Boolean> added = optionalValue.map(results::add);
```

Now added has one of three values: true or false wrapped into an Optional if optionalValue was present, or an empty Optional otherwise.

8.7.2 HOW NOT TO WORK WITH OPTIONAL VALUES

Misusing Optional values leads to no benefits over the "something or null" approach of the past.

The **get method** retrieves the wrapped element of an Optional value if it exists, or throws a NoSuchElementException if it doesn't. Therefore, calling:

```
Optional<T> optionalValue = ...;
optionalValue.get().someMethod();
```

is no safer than:

```
T value = ...;
value.someMethod();
```

Similarly, the **isPresent** method checks whether an Optional<T> object has a value, but using it as follows:

```
if (optionalValue.isPresent())
optionalValue.get().someMethod();
```

offers no simplicity over:

```
if (value != null) value.someMethod();
```

8.7.3 CREATING OPTIONAL VALUES

If you need to create an Optional object yourself, there are several static methods available for that purpose, including **Optional.of(result)** and **Optional.empty()**. For instance:

```
public static Optional<Double> inverse(Double x){
  return x == 0 ? Optional.empty():Optional.of(1 / x);
}
```

The **ofNullable** method serves as a bridge from possibly null values to optional values. Optional.ofNullable(obj) returns Optional.of(obj) if obj is not null and Optional.empty() otherwise.

8.7.4 COMPOSING OPTIONAL VALUE FUNCTIONS WITH FLATMAP

Suppose you have a method f yielding an Optional<T>, and the target type T has a method g yielding an Optional<U>. If they were normal methods, you could compose them by calling s.f().g(). However, this **composition doesn't work since s.f() has type Optional<T>, not T**. Instead, call:

```
Optional<U> result = s.f().flatMap(T::g);
```

If s.f() is present, then g is applied to it. Otherwise, an empty Optional<U> is returned.

You can **repeat this process if you have more methods or lambdas that yield Optional values**, enabling you to build a pipeline of steps by **chaining calls to flatMap**. For example, suppose you have a safe inverse method and a safe square root method:

```
public static Optional<Double> squareRoot(Double x){
  return x < 0 ? Optional.empty():Optional.of(Math.sqrt(x));
}
```

Then you can compute the square root of the inverse as:

```
Optional<Double> result = inverse(x).flatMap(MyMath::squareRoot);
```

or, alternatively:

```
Optional<Double> result = Optional.of(-4.0).flatMap(Demo::inverse).flatMap(Demo::squareRoot);
```

If either the inverse method or the squareRoot method returns Optional.empty(), the result is empty.

8.8 COLLECTING RESULTS

Once you've processed a stream, you'll often want to examine the results. There are several methods available for this purpose.

You can **call the iterate method to obtain an old-fashioned iterator that allows you to visit the elements**.

Alternatively, **use the forEach method to apply a function to each element**:

```
stream.forEach(System.out::println);
```

On a parallel stream, the forEach method traverses elements in arbitrary order. If you need to process them in stream order, call **forEachOrdered** instead, though this may sacrifice some parallelism benefits.

More commonly, you'll want to collect the results into a data structure. You can **use the toArray method to**

obtain an array of the stream elements. Since it's not possible to create a generic array at runtime, stream.toArray() returns an Object[] array. If you need an array of the correct type, pass in the array constructor:

```
String[] result = stream.toArray(String[]::new);
```

stream.toArray() has type Object[]

For collecting stream elements into another target, use the convenient collect method, which takes an instance of the Collector interface. The Collectors class provides numerous factory methods for common collectors. To collect a stream into a list or set, simply call:

```
List<String> result =
stream.collect(Collectors.toList());
// or
Set<String> result =
stream.collect(Collectors.toSet());
```

If you want to **control the type of set you get, use the following call instead**:

```
TreeSet<String> result =
stream.collect(Collectors.toCollection(TreeSet:
:new));
```

8.9 COLLECTING INTO MAPS

When you have a Stream<Person> and want to collect the elements into a map for later lookup by their ID, you can utilize the **Collectors.toMap method**. This method requires two function arguments that produce the map's keys and values. For instance:

```
Map<Integer, String> idToName =
people.collect(Collectors.toMap(Person::getId,
Person::getName));
```

In the common case when the values should be the actual elements, use **Function.identity()** for the second function:

```
Map<Integer, Person> idToPerson =
people.collect(Collectors.toMap(Person::getId,
Function.identity()));
```

If there are multiple elements with the same key, a conflict occurs, and the collector throws an **IllegalStateException**. You can customize this behavior by supplying a third function argument that resolves the conflict and determines the value for the key based on the existing and new values.

If you desire a **TreeMap**, supply its constructor as the fourth argument along with a merge function. Here's an example:

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(
        Person::getId,
        Function.identity(),
        (existingValue, newValue) -> { throw
        new IllegalStateException(); },
        TreeMap::new
    )
);
```

8.10 GROUPING AND PARTITIONING

The process involved creating singleton sets and specifying how to merge existing and new values, can be somewhat cumbersome.

The **groupingBy** method directly supports forming groups of values with the same characteristic. Let's take the example of grouping locales by country. We can achieve this by using the groupingBy collector with the classifier function Locale::getCountry. This function categorizes locales by their country code:

```
Map<String, List<Locale>> countryToLocales =
locales.collect(
    Collectors.groupingBy(Locale::getCountry));
```

Now, you can easily look up all locales for a given country code:

```
List<Locale> swissLocales =
countryToLocales.get("CH");
// Yields locales [it_CH, de_CH, fr_CH]
```

In cases where the classifier function is a predicate function (returns a boolean value), and you want to partition the stream elements into two lists based on whether the function returns true or false, using **partitioningBy** is more efficient than groupingBy.

For instance, if we want to split all locales into those that use English and all others:

```
Map<Boolean, List<Locale>>
englishAndOtherLocales = locales.collect(
    Collectors.partitioningBy(l ->
    l.getLanguage().equals("en")));
List<Locale> englishLocales =
englishAndOtherLocales.get(true);
```

This creates a map where the boolean keys represent whether the locale's language is English or not, and the corresponding values are lists of locales.

Additionally, if you're working with parallel streams and need a concurrent map, you can use the **groupingByConcurrent** method, which behaves similarly to the toConcurrentMap method.

```
Map<String, List<Locale>> countryToLocales =
locales.collect(
    Collectors.groupingBy(Locale::getCountry)
);
```

Here, the function Locale::getCountry serves as the classifier function for grouping. Now you can easily look up all locales for a given country code. For example:

```
List<Locale> swissLocales =
countryToLocales.get("CH");
```

Yields locales [it_CH, de_CH, fr_CH]

8.11 DOWNSTREAM COLLECTORS

The **groupingBy** method yields a map whose values are lists. If you want to process those lists in some way, you can supply a downstream collector. For example, if you

want sets instead of lists, you can use the **Collectors.toSet** collector:

```
Map<String, Set<Locale>> countryToLocaleSet =
locales.collect(
    groupingBy(Locale::getCountry, toSet())
);
```

Several collectors are provided for reducing grouped elements to numbers:

counting produces a count of the collected elements:

```
Map<String, Long> countryToLocaleCounts =
locales.collect(
    groupingBy(Locale::getCountry, counting())
);
```

summing(Int|Long|Double) takes a function argument, applies the function to the downstream elements, and produces their sum:

```
Map<String, Integer> stateToCityPopulation =
cities.collect(
    groupingBy(City::getState, summingInt(City:
                :getPopulation))
);
```

maxBy and **minBy** take a comparator and produce maximum and minimum of the downstream elements:

```
Map<String, City> stateToLargestCity =
cities.collect(
    groupingBy(City::getState, maxBy(Comparator
               .comparing(City::getPopulation)))
);
```

mapping applies a function to downstream results, and it requires yet another collector for processing its results:

```
Map<String, Optional<String>>
stateToLongestCityName = cities.collect(
    groupingBy(City::getState, mapping(City::ge
                tName, maxBy(Comparator.compari
                ng(String::length))))
);
```

Here, we group cities by state. Within each state, we produce the names of the cities and reduce by maximum length.

The **mapping method** also yields a nicer solution to a problem from the preceding section—gathering a set of all languages in a country:

```
Map<String, Set<String>> countryToLanguages =
locales.collect(
   groupingBy(Locale::getDisplayCountry,
   mapping(Locale::getDisplayLanguage,toSet()))
);
```

8.12 REDUCTION OPERATIONS

The **reduce** method is a general **mechanism for computing a value from a stream**. The simplest form takes a binary function and keeps applying it, starting with the first two elements. It returns an Optional because there is no valid result if the stream is empty.

```
List<Integer> values = ...;
Optional<Integer> sum =
values.stream().reduce((x, y) -> x + y);
```

In general, if the reduce method has a reduction operation op, the reduction yields u0 op u1 op u2 op ..., where ui op ui + 1 denotes the function call op(ui, ui + 1). **The operation should be associative**: It shouldn't matter in which order you combine the elements.

In practice, you might not use the reduce method frequently. It's usually easier to map to a stream of numbers and use one of its methods to compute sum, max, or min. Additionally, when dealing with complex data types, reduce might not be general enough, and you might need to use collect instead.

8.13 PRIMITIVE TYPE STREAMS

In handling numerical data in streams, it's inefficient to wrap each value into a wrapper object. To address this, Java offers **specialized streams for primitive types** like int, long, and double, namely **IntStream**, **LongStream**, and **DoubleStream**, respectively. For other primitive types such as short, char, byte, and boolean, you can utilize **IntStream**, and for float, you'd use **DoubleStream**.

Creating an IntStream can be done using methods like **IntStream.of** and **Arrays.stream**:

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
stream = Arrays.stream(values, from, to);
// values is an int[] array
```

When dealing with characters in a string, you can access their Unicode codes using methods like codePoints:

```
String sentence = "\uD835\uDD46 is the set of
octonions."; // UTF-16 encoding of a letter
IntStream codes = sentence.codePoints(); // The
stream with hex values 1D546 20 69 73 20 ...
```

Converting a stream of objects to a primitive type stream and vice versa is achievable using methods like **mapToInt**, **mapToLong**, **mapToDouble**, and **boxed**:

```
Stream<String> words = …;
IntStream lengths =
words.mapToInt(String::length);
Stream<Integer> integers = IntStream.range(0,
100).boxed();
```

Notable differences between primitive type streams and object streams include:

- The toArray methods return primitive type arrays.
- Methods that yield an optional result return an OptionalInt, OptionalLong, or OptionalDouble. These classes are analogous to the Optional class, but they have methods getAsInt, getAsLong, and getAsDouble instead of the get method.
- There are methods sum, average, max, and min that return the sum, average, maximum, and minimum. These methods are not defined for object streams.
- The summaryStatistics method yields an object of type IntSummaryStatistics, LongSummaryStatistics, or

DoubleSummaryStatistics that can simultaneously report the sum, average, maximum, and minimum of the stream.

Note

Additionally, the Random class offers methods like ints, longs, and doubles to generate primitive type streams of random numbers.

8.14 PARALLEL STREAMS

In Java, **streams offer a straightforward way to parallelize bulk operations**.

However, to ensure correct behavior, certain guidelines must be followed. First and foremost, you need to ensure that you're **working with a parallel stream**. You can obtain a parallel stream from any collection using the **parallelStream()** method:

```
Stream<String> parallelWords =
words.parallelStream();
```

Additionally, the parallel() method can convert any sequential stream into a parallel one:

```
Stream<String> parallelWords =
Stream.of(wordArray).parallel();
```

When executing terminal operations on a parallel stream, all intermediate stream operations are automatically parallelized. However, it's crucial to note that parallel execution doesn't guarantee the same result as serial execution. Therefore, operations must be stateless and capable of being executed in any order.

Consider an example where we attempt to count short words in a stream of strings:

```
int[] shortWords = new int[12];
words.parallelStream().forEach(
    s -> { if (s.length() < 12)
    shortWords[s.length()]++; });
// Error—race condition!
System.out.println(Arrays.toString(shortWords));
```

This code snippet is problematic as the function passed to forEach runs concurrently in multiple threads, each updating a shared array, leading to a classic race condition.

To ensure safe parallel execution, functions passed to parallel stream operations should avoid mutable state. For instance, you can safely parallelize the computation by grouping strings by length and counting them:

```
Map<Integer, Long> shortWordCounts =
    words.parallelStream()
          .filter(s -> s.length() < 10)
          .collect(groupingBy(String::length,
            counting()));
```

By default, **streams from ordered collections maintain their order throughout operations**. However, **ordering doesn't hinder efficient parallelization**. Operations like map can partition the stream into segments, process them concurrently, and reassemble the results in order.

Some operations, like distinct, can benefit from dropping ordering, enhancing parallelization. Similarly, calling unordered() before limit() speeds up the process by removing ordering constraints.

When collecting results into maps with parallel streams, note that **the order of map values may not match the stream order due to merging maps being expensive**. This is evident when using groupingByConcurrent. However, if the downstream collector is independent of ordering, this discrepancy may not matter.

Finally, it's important not to modify the collection backing a stream while executing stream operations, as this violates the noninterference principle. Although intermediate stream operations are lazy, mutations to the collection before terminal operations execute can lead to undefined behavior.

For example, the following is incorrect:

```
Stream<String> words = wordList.stream();
words.forEach(s -> if (s.length() < 12)
wordList.remove(s)); // Error—interference
```

CONCURRENT PROGRAMMING

Java was one of the first mainstream programming languages with built-in support for concurrent programming.

9.1 CONCURRENT TASKS

When you design a concurrent program, you need to think about the **tasks that can be run in parallel**.

9.1.1 RUNNING TASKS

In Java, the **Runnable interface describes a task you want to run**, usually concurrently with others.

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

The code of the run method will be executed in a thread.

A **thread** is a **mechanism for execution of a sequence of instructions**, usually provided by the operating system. Multiple **threads run concurrently**, by using separate processors or different time slices on the same processor.

In practice, it doesn't usually make sense to have a one-to-one relationship between tasks and threads. When tasks are short lived, you want to **run many of them on the same thread**, so you don't waste the time it takes to start a thread. When your tasks are computationally intensive, you just want one thread per processor instead of one thread per task, to avoid the overhead of switching among threads.

In the Java concurrency library, an executor executes tasks, choosing the threads on which to run them.

```
Runnable task = () -> { ... };
Executor exec = ...;
exec.execute(task);
```

The **Executors** class has factory methods for different types of executors. The call

```
exec = Executors.newCachedThreadPool();
```

yields an executor optimized for programs with many tasks that are short lived or spend most of their time waiting. Each task is executed on an idle thread if possible, but a new thread is allocated if all threads are busy. Threads that are idle for an extended time are terminated.

The call

```
exec = Executors.newFixedThreadPool(nthreads);
```

creates a pool with a fixed number of threads. When you give it a task, it waits in a queue until a thread is available to work on it. This is good for tasks that need a lot of computing power. You can decide how many threads to have based on how many processors your computer has, which you can find out using

```
int processors =
Runtime.getRuntime().availableProcessors();
```

Esempio:

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class Main {

    public static void main(String[] args) {
        // Creiamo un executor con un pool di
           thread fissato a 3
        Executor exec =
              Executors.newFixedThreadPool(3);

        // Definiamo un array di compiti
        Runnable[] tasks = {

            () -> {
               int sum = 0;
               for (int i = 1; i <= 100; i++) {
                    sum += i;
               }
               System.out.println("La somma dei
                primi 100 numeri è: " + sum);
            },

            () -> {
              int sum = 0;
              for (int i = 101; i <= 200; i++) {
                    sum += i;
              }
              System.out.println("La somma dei
               numeri da 101 a 200 è: " + sum);
            },

            () -> {
              int sum = 0;
              for (int i = 201; i <= 300; i++) {
                    sum += i;
              }
              System.out.println("La somma dei
               numeri da 201 a 300 è: " + sum);
            }
        };

        // Eseguiamo i compiti utilizzando
           l'executor
        for (Runnable task : tasks) {
            exec .execute(task);
        }
    }
}
```

In questo esempio, abbiamo creato un executor con un pool di thread fissato a 3. Abbiamo definito una serie di compiti da eseguire, ognuno dei quali calcola la somma di un insieme di numeri. Utilizzando l'executor, eseguiamo i compiti in parallelo. L'executor si occuperà di assegnare i compiti ai thread disponibili nel pool e di gestire la loro esecuzione in modo efficiente.

9.1.2 FUTURES AND EXECUTOR SERVICES

Consider a computation that splits into multiple subtasks, each of which computes a partial result. When all tasks are done, you want to combine the results. You can use the **Callable interface** for the subtasks. Its call method,

unlike the run method of the Runnable interface, returns a value:

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

As a bonus, the call method can throw arbitrary exceptions.

To execute a Callable, **you need an instance of the ExecutorService interface**, a subinterface of Executor. The **newCachedThreadPool** and **newFixedThreadPool** methods of the Executors class yield such objects.

```
ExecutorService exec =
Executors.newFixedThreadPool();
Callable<V> task = ...;
Future<V> result = exec.submit(task);
```

When you submit the task, you get a **future**—an object that represents a computation whose result will be available at some future time. The Future interface has the **following methods**:

```
V result = future.get();
```

Restituisce il risultato del compito, bloccando fino a quando non è disponibile.

```
V resultWithTimeout = future.get(long timeout,
TimeUnit unit);
```

Restituisce il risultato con un timeout

```
boolean wasCancelled = future.cancel(boolean
mayInterruptIfRunning);
```

Cancella il compito, eventualmente interrompendo l'esecuzione

```
boolean isCancelled = future.isCancelled();
```

Verifica se il compito è stato cancellato

```
boolean isDone = future.isDone();
```

Verifica se il compito è completato

Usually, a task needs to wait for the result of multiple subtasks. Instead of submitting each subtask separately, you can use the **invokeAll method**, passing a Collection of Callable instances.

For example, suppose you want to count how often a word occurs in a set of files. For each file, make a Callable<Integer> that returns the count for that file. Then submit them all to the executor. When all tasks have completed, you get a list of the futures (all of which are done), and you can total up the answers.

```
String word = ...;
Set<Path> paths = ...;
List<Callable<Long>> tasks = new ArrayList<>();
for (Path p : paths) tasks.add(
   () -> { return number of occurrences of word
           in p });
List<Future<Long>> results =
executor.invokeAll(tasks); long total = 0;
for (Future<Long> result : results) total +=
result.get();
```

There is also a variant of invokeAll with a timeout, which cancels all tasks that have not completed when the timeout is reached.

The **invokeAny method** is like invokeAll, but it returns as soon as any one of the submitted tasks has completed without throwing an exception. It then returns the value of its Future. The other tasks are cancelled. This is useful for a search that can conclude as soon as a match has been found.

9.2 THREAD SAFETY

9.2.1 VISIBILITY

Even operations as simple as writing and reading a variable can be incredibly complicated with modern processors. Consider this example:

```
public class Main {
   private static boolean done = false;
   public static void main(String[] args) {
      Runnable hellos = () -> {
         for (int i = 1; i <= 1000; i++)
            System.out.println("Hello " + i);
         done = true;
        };
      Runnable goodbye = () -> {
          int i = 1;
          while (!done)
             i++;
          System.out.println("Goodbye " + i);
      };
   Executor executor = Executors.newCachedThrea
                                        dPool();
   executor.execute(hellos);
   executor.execute(goodbye);
    }
}
```

The first task prints "Hello" a thousand times, and then sets done to true. The second task waits for done to become true, and then prints "Goodbye" once, incrementing a counter while it is waiting for that happy moment.

You'd expect the output to be something like

```
Hello 1
...
Hello 1000
Goodbye 501249
```

When you run this program on your laptop, the program prints up to "Hello 1000" and never terminates. The effect of **done = true is not visible** to the thread running the second task.

Why? A processor tries to hold the data that it needs in registers or an onboard memory cache, and eventually writes changes back to memory for processor performance purposes. There are operations for synchronizing cached copies, but they are only issued when requested.

Moreover, the compiler, the virtual machine, and the processor are allowed to change the order of instructions to speed up operations, provided it does not change the semantics of the program.

By default, **optimizations assume that there are no concurrent memory accesses**. If there are, the virtual machine needs to know, so that it can then emit processor instructions that inhibit improper reorderings.

There are several **ways of ensuring that an update to a variable is visible**. Here is a summary:

- The value of a **final variable** is visible after initialization.
- The **initial value of a static variable** is visible after static initialization.
- Changes to a **volatile variable** are visible.
- Changes that happen before **releasing a lock** are visible to anyone acquiring the same lock

In our case, the problem goes away if you declare the shared variable done with the volatile modifier:

```
private static volatile boolean done;
```

Tip

It is an excellent idea to declare any field that does not change after initialization as final. Then you never have to worry about its visibility.

9.2.2 RACE CONDITIONS

Suppose multiple concurrent tasks update a shared integer counter.

```
private static volatile int count = 0;
...
count++; // Task 1
...
count++; // Task 2
...
```

The variable has been declared as volatile, so the updates are visible. But that is not enough.

The update count++ is not atomic. It actually means

```
count = count + 1;
```

and it can be interrupted if a thread is preempted before it stores the value count + 1 back into the count variable. Consider this scenario:

```
int count = 0; // Initial value
register1 = count + 1; // Thread 1 computes
                          count + 1
... // Thread 1 is preempted
register2 = count + 1; // Thread 2 computes
                          count + 1
count = register2; // Thread 2 stores 1 in count
... // Thread 1 is running again
count = register1; // Thread 1 stores 1 in count
```

Now count is 1, not 2. This kind of error is called a **race condition** because it **depends on which thread wins the "race" for updating the shared variable**.

Race conditions are a problem whenever shared variables are mutated. For example, when adding a value to the head of a queue, the insertion code might look like this:

```
Node n = new Node();
if (head == null) head = n;
else tail.next = n;
tail = n;
tail.value = newValue;
```

Lots of things can go wrong if this complex sequence of instructions is paused at an unfortunate time and another task gets control, accessing the queue while it is in an inconsistent state.

There is a remedy to this problem: **Use locks to make critical sequences of operation atomic**.

9.2.3 STRATEGIES FOR SAFE CONCURRENCY

When it comes to accessing shared data in concurrent programs, Java lacks a mechanism for automatic management. As a result, programmers must adhere to a set of guidelines to mitigate the risks associated with concurrent access to shared data.

A highly effective strategy is **confinement**. Just say no when it comes to sharing data among tasks.

Another good strategy is **immutability**. It is safe to share immutable objects.

The third strategy is **locking**. By granting only one task at a time access to a data structure, one can keep it from being damaged. Locking can be expensive since it **reduces opportunities for parallelism**. If most tasks have to wait their turn, they aren't doing useful work.

9.2.4 IMMUTABLE CLASSES

A class is immutable when **its instances, once constructed, cannot change**.

It is not difficult to implement immutable classes, but you should pay attention to these issues:

- Be sure to **declare instance variables final**. There is no reason not to, and you gain an important advantage. The virtual machine ensures that a final instance variable is visible after construction.
- **None of the methods can be mutators**. You may want to make them final, or declare the class final, so that methods can't be overridden with mutators.
- **Don't leak mutable state**. None of your (non-private) methods can return a reference to any innards that could be used for mutation. Also, when one of your methods calls a method of another class, it must not pass any such references, since the called method might otherwise use it for mutation. If necessary, return or pass a copy.
- **Don't let the this reference escape in a constructor**. When you call another method, you know not to pass any internal references. If you revealed this in the constructor, someone could observe the object in an incomplete state.

9.3 PARALLEL ALGORITHMS

9.3.1 PARALLEL STREAMS

The stream library automatically parallelizes operations on large parallel streams. For example, if coll is a large collection of strings, and you want to find how many of them start with the letter A, call

```
long result = coll.parallelStream().filter(s -> s.startsWith("A")).count();
```

The parallelStream method yields a parallel stream. The stream is broken up into segments. The filtering and counting is done on each segment, and the results are combined. You don't need to worry about the details.

Caution

When you use parallel streams with lambdas (for example, as the argument to filter in the preceding example), be sure to **stay away from unsafe mutation of shared objects**.

9.3.2 PARALLEL ARRAY OPERATIONS

The Arrays class has a number of **parallelized operations**. Just as with the parallel stream operations of the preceding sections, the operations **break the array into sections, work on them in parallel, and combine the results**.

The **static Arrays.parallelSetAll method** fills an array with values computed by a function. The function receives the element index and computes the value at that location.

```
Arrays.parallelSetAll(values, i -> i % 10);
// Fills values with 0123456789012 ...
```

Clearly, this operation benefits from being parallelized. There are versions for all primitive type arrays and for object arrays.

The **parallelSort method** can sort an array of primitive values or objects. For example,

```
Arrays.parallelSort(words,
Comparator.comparing(String::length));
```

With all methods, you can supply the bounds of a range, such as

```
Arrays.parallelSort(values, values.length / 2, values.length); // Sort the upper half
```

9.4 THREADSAFE DATA STRUCTURES

When multiple threads concurrently modify a data structure like a queue or hash table, there's a risk of corrupting its internal state. Java's **java.util.concurrent** package provides collections designed to handle concurrent access without blocking threads, ensuring thread safety

9.4.1 CONCURRENT HASH MAPS

A **ConcurrentHashMap** is, first of all, a hash map whose operations are threadsafe. No matter how many threads operate on the map at the same time, the internals are not corrupted. Of course, **some threads may be temporarily blocked**, but the map can efficiently support a large number of concurrent readers and a certain number of concurrent writers.

To update a value safely, use the **compute** method. It is called with a key and a function to compute the new value. That function receives the key and the associated value, or null if there is none, and computes the new value. The compute method is **atomic**—no other thread can mutate the map entry while the computation is in progress.

There are also variants **computeIfPresent** and **computeIfAbsent** that only compute a new value when there is already an old one, or when there isn't yet one.
Another atomic operation is **putIfAbsent**. You often need to do something special when a key is added for the first time. The **merge** method makes this particularly convenient. It has a parameter for the initial value that is used when the key is not yet present. Otherwise, the function that you supplied is called, combining the existing value and the initial value. (Unlike compute, the function does not process the key.)

9.4.2 BLOCKING QUEUES

One commonly used tool for coordinating work between tasks is a **blocking queue**. Producer tasks insert items into the queue, and consumer tasks retrieve them. The queue **lets you safely hand over data from one task to another**.

When you try to add an element and the queue is currently full, or you try to remove an element when the queue is empty, the operation blocks.

The blocking queue methods fall into three categories that differ by the action they perform when the queue is full or empty. In addition to the blocking methods, there are methods that throw an exception when they don't succeed, and methods that return with a failure indicator instead of throwing an exception if they cannot carry out their tasks.

Method	Normal Action
put	Adds an element to the tail
take	Removes and returns the head element
add	Adds an element to the tail
remove	Removes and returns the head element
element	Returns the head element
offer	Adds an element and returns true
poll	Removes and returns the head element
peek	Returns the head element

A **LinkedBlockingQueue** is based on a linked list, and an ArrayBlockingQueue uses a circular array.
A common challenge with such a design is stopping the consumers. A consumer cannot simply quit when the queue is empty. After all, the producer might not yet have started, or it may have fallen behind. If there is a single producer, it can add a "last item" indicator to the queue, similar to a dummy suitcase with a label "last bag" in a baggage claim belt.

9.4.3 OTHER THREADSAFE DATA STRUCTURES

In **java.util.concurrent** package, alongside hash maps and tree maps, there's a concurrent map named **ConcurrentSkipListMap**. It's useful for traversing keys in sorted order or accessing methods from the NavigableMap interface.

Similarly, **CopyOnWriteArrayList** and **CopyOnWriteArraySet** are thread-safe collections where mutators create a copy of the underlying array.
This setup is beneficial when iteration frequency outweighs mutation frequency, providing consistent access without synchronization overhead.

If you need a large thread-safe set, you can use **ConcurrentHashMap.newKeySet()**, which returns a Set<K> wrapping a ConcurrentHashMap<K, Boolean>. Although values are Boolean.TRUE, they are not used, making it practical for set operations.

Additionally, the **keySet** method of an existing map yields a mutable set of keys. Removing elements from this set also removes their corresponding keys and values from the map. However, adding elements to this set doesn't make sense, so a second keySet method with a default value is provided for adding elements.

9.5 ATOMIC VALUES

If multiple threads update a shared counter, you need to make sure that this is done in a threadsafe way. There are a number of classes in the **java.util.concurrent.atomic** package that use safe and efficient machine-level instructions to guarantee **atomicity of operations** on integers, long and boolean values, object references, and arrays thereof.
For example, you can safely generate a sequence of numbers like this:

```
public static AtomicLong nextNumber = new
AtomicLong(); // In some thread...
long id = nextNumber.incrementAndGet();
```

The **incrementAndGet** method atomically increments the AtomicLong and returns the post-increment value. That is, the operations of getting the value, adding 1, setting it, and producing the new value cannot be interrupted. It is **guaranteed that the correct value is computed and returned**, even if multiple threads access the same instance concurrently.

If you want to make a more complex update. One way is to use the **updateAndGet method**. For example, suppose you want to keep track of the largest value that is observed by different threads, call updateAndGet with a lambda expression for updating the variable. In our example, we can call

```
largest.updateAndGet(x -> Math.max(x, observed));
```

or

```
largest.accumulateAndGet(observed, Math::max);
```

The **accumulateAndGet method** takes a binary operator that is used to combine the atomic value and the supplied argument.
There are also methods **getAndUpdate** and **getAndAccumulate** that return the old value.

When you have a **very large number of threads accessing the same atomic values, performance suffers** because updates are carried out optimistically. That is, **the operation computes a new value from a given old value**, then **does the replacement provided the old value is still the current one, or retries if it is not**. Under heavy contention, updates require too many retries.

The classes **LongAdder** and **LongAccumulator** solve this problem for certain common updates. A LongAdder is composed of multiple variables whose collective sum is the current value. Multiple threads can update different summands, and new summands are automatically provided when the number of threads increases. This is efficient in the common situation where the value of the sum is not needed until after all work has been done.

If you anticipate high contention, you should simply use a **LongAdder** instead of an **AtomicLong**. The method names are slightly different. Call increment to increment a counter or add to add a quantity, and sum to retrieve the total.

```
final LongAdder count = new LongAdder();
for (...)
    executor.execute(() -> {
        while (...) {
            ...
            if (...) count.increment();
        }
    });
...
long total = count.sum();
```

The **LongAccumulator** generalizes this idea to an arbitrary accumulation operation. In the constructor, you provide the operation as well as its neutral element. To incorporate new values, call accumulate. Call get to obtain the current value.

```
LongAccumulator accumulator = new
LongAccumulator(Long::sum, 0);  // In some tasks…
accumulator.accumulate(value);
// When all work is done
long sum = accumulator.get();
```

Internally, **the accumulator has variables a1, a2, ..., an**. Each variable is initialized with the neutral element (0 in our example).

When accumulate is called with value u, then one of them is atomically updated as ai = ai **op** u, where op is the accumulation operation written in infix form. In our example, a call to accumulate computes ai = ai + u for some i.

The result of get is a1 op a2 op ... op an. In our example, that is the sum of the accumulators, a1 + a2 + ... + an.

If you choose a different operation, it **must be associative and commutative**. That means that the final result must be independent of the order in which the intermediate values were combined.

There are also **DoubleAdder** and **DoubleAccumulator** that work in the same way, except with double values.

9.6 LOCKS

Now we look at how one would build a threadsafe counter or blocking queue.

9.6.1 REENTRANT LOCKS

To avoid the corruption of shared variables, one needs to ensure that only one thread at a time can compute and set the new values. **Code that must be executed in its entirety, without interruption**, is called a **critical section**. One can use a lock to implement a critical section:

```
Lock countLock = new ReentrantLock();
// Shared among multiple threads
int count;
// Shared among multiple threads
...
countLock.lock();
try {
    count++; // Critical section
} finally {
   countLock.unlock();
   // Make sure the lock is unlocked
}
```

The first thread to execute the lock method locks the countLock object and then proceeds into the critical section. **If another thread tries to call lock on the same object, it is blocked until the first thread executes the call to unlock**. In this way, it is guaranteed that only one thread at a time can execute the critical section.

Note that, by placing the unlock method into a finally clause, the lock is released if any exception happens in the critical section. Otherwise, the lock would be permanently locked, and no other thread would be able to proceed past it. This would clearly be very bad. Of course, in this case, the critical section can't throw an exception since it only executes an integer increment. But it is a common idiom to use the try/finally statement anyway, in case more code gets added later.

Although using locks can look easy, it is just as easy to use the wrong locks, or create situations that deadlock when no thread can make progress because all of them wait for a lock.

For that reason, application programmers should use locks as a matter of last resort. First **try to avoid sharing**, by using immutable data or handing off mutable data from one thread to another. If you must share, **use prebuilt threadsafe structures such as a ConcurrentHashMap or a LongAdde**r. It is useful to know about locks so you can understand how such data structures can be implemented, but it is best to leave the details to the experts.

9.6.2 THE SYNCHRONIZED KEYWORD

In the preceding section, we used a ReentrantLock to implement a critical section. You don't have to use an explicit lock because in Java, every object has an intrinsic lock. To understand intrinsic locks, however, it helps to have seen explicit locks first.

The synchronized keyword is used to lock the intrinsic lock. It can occur in two forms. You can lock a block:

```
synchronized (obj) {
     Critical section
}
```

This essentially means

```
obj.intrinsicLock.lock(); try {
   Critical section
} finally {
   obj.intrinsicLock.unlock();
}
```

An object does not actually have a field that is an intrinsic lock. The code is just meant to *illustrate* what goes on when you use the synchronized keyword.

You can also **declare a method as synchronized**. Then its body is locked on the receiver parameter this. That is,

```
public synchronized void method() {
   Body
}
```

is the equivalent of

```
public void method() {
   this.intrinsicLock.lock();
   try {
      Body
   } finally {
      this.intrinsicLock.unlock();
   }
}
```

For example, a counter can simply be declared as

```
public class Counter {
   private int value;
   public synchronized int increment() {
      value++;
      return value;
   }
}
```

By using the intrinsic lock of the Counter instance, there is **no need to come up with an explicit lock**.

Synchronized methods were inspired by the monitor concept. A **monitor** is essentially **a class in which all instance variables are private and all methods are protected by a private lock**.

In Java, it is possible to have public instance variables and to mix synchronized and unsynchronized methods. More problematically, the intrinsic lock is publicly accessible.

9.6.3 WAITING ON CONDITIONS

Consider a simple Queue class with methods for adding and removing objects. Synchronizing the methods ensures that these operations are atomic.

```
public class Queue {
   class Node { Object value; Node next; };
   private Node head;
   private Node tail;
   public synchronized void add(Object newValue){
      Node n = new Node();
      if (head == null) head = n;
      else tail.next = n;
      tail = n;
      tail.value = newValue;
   }
   public synchronized Object remove() {
      if (head == null) return null;
      Node n = head;
      head = n.next;
      return n.value;
   }
}
```

Now suppose we want to turn the remove method into a method take that blocks if the queue is empty.

The **check for emptiness must come inside the synchronized method because otherwise the inquiry would be meaningless**—another thread might have emptied the queue in the meantime.

```
public synchronized Object take() {
   if (head == null) ... // Now what?
   Node n = head;
   head = n.next;
   return n.value;
   }
```

But what should happen if the queue is empty? No other thread can add elements while the current thread holds the lock. This is where the **wait method** comes in.

If the take method finds that it cannot proceed, it calls the wait method:

```
public synchronized Object take() throws
InterruptedException {
   while (head == null) wait();
   ...
}
```

The current thread is now deactivated and gives up the lock. This lets in another thread that can, we hope, add elements to the queue. This is called waiting on a condition.

Note

That the wait method is a method of the Object class. It relates to the lock that is associated with the object.

There is an essential difference between a **thread that is blocking to acquire a lock** and a **thread that has called wait**.

Once a thread calls the wait method, it enters a **wait set** for the object. The **thread is not made runnable** when the lock is available. Instead, it stays deactivated until another thread has called the notifyAll method on the same object.

When another thread has added an element, it should call that method:

```
public synchronized void add(Object newValue) {
  ...
  notifyAll();
}
```

The call to notifyAll reactivates all threads in the wait set. **When the threads are removed from the wait set, they are again runnable and the scheduler will eventually activate them again**. At that time, they will attempt to reacquire the lock. As one of them succeeds, it continues where it left off, returning from the call to wait.

At this time, the thread should test the condition again. There is no guarantee that the condition is now fulfilled —the notifyAll method merely signals to the waiting threads that it may be fulfilled at this time and that it is worth checking for the condition again.

For that reason, the test is in a loop

```
while (head == null) wait();
```

Caution

Another method, notify, unblocks only a single thread from the wait set. That is more efficient than unblocking all threads, but there is a danger. If the chosen thread finds that it still cannot proceed, it becomes blocked again. If no other thread calls notify again, the program deadlocks.

A thread can only call **wait**, **notifyAll**, or **notify** on an object if it holds the lock on that object.

9.7 THREADS

Threads are the **primitives that actually execute tasks**. Normally, you are better off using executors that manage threads for you, but the following sections give you some background information about working directly with threads.

9.7.1 STARTING A THREAD HERE IS HOW TO RUN A THREAD IN JAVA.

```
Runnable task = () -> { ... };
Thread thread = new Thread(task);
thread.start();
```

The static sleep method makes the current thread sleep for a given period, so that some other threads have a chance to do work.

```
Runnable task = () -> {
   …
   Thread.sleep(millis);
   …
}
```

If you want to wait for a thread to finish, call the join method:

```
thread.join(millis);
```

A **thread ends when its run method returns**, either normally or because an exception was thrown. In the latter case, the uncaught exception handler of the thread is invoked. When the thread is created, that handler is set to the uncaught exception handler of the thread group, which is ultimately the global handler.

9.7.2 THREAD INTERRUPTION

Suppose that, for a given query, you are always satisfied with the first result. When the search for an answer is distributed over multiple tasks, you want to cancel all others as soon as the answer is obtained. In Java, **task cancellation is cooperative**.

Each thread has an **interrupted status** that indicates that someone would like to "interrupt" the thread.

A Runnable can check for this status, which is typically done in a loop:

```
Runnable task = () -> {
   while (more work to do) {
     if(Thread.currentThread().isInterrupted())
                                        return;
     Do more work
   }
};
```

When the thread is interrupted, the run method simply ends.

Sometimes, a thread becomes **temporarily inactive**. That can happen if a thread waits for a value to be computed by another thread or for input/output, or if it goes to sleep to give other threads a chance.
If the thread is interrupted while it waits or sleeps, it is immediately reactivated—but in this case, the interrupted status is not set. Instead, an **InterruptedException** is thrown. This is a checked exception, and you must catch it inside the run method of a Runnable. The usual reaction to the exception is to end the run method:

```
Runnable task = () -> {
   try {
      while (more work to do) {
         Do more work
         Thread.sleep(millis);
      }
   }
   catch (InterruptedException ex) {
      // Do nothing }
};
```

When you catch the InterruptedException in this way, there is **no need to check for the interrupted status**. If the thread was interrupted outside the call to Thread.sleep, the status is set and the Thread.sleep method throws an InterruptedException as soon as it is called.

9.7.3 THREAD-LOCAL VARIABLES

Thread-local variables provide a solution to the problem of thread safety when multiple threads need access to an object that is not inherently thread-safe. Instead of sharing a single instance of the object among all threads, **each thread gets its own separate instance**.

For example, consider the NumberFormat class, which is not thread-safe. If multiple threads attempt to use a shared NumberFormat instance concurrently, it can lead to corrupted results.
To avoid this, you can use the ThreadLocal helper class to create a separate NumberFormat instance for each thread. This ensures that each thread has its own isolated instance, eliminating the risk of corruption due to concurrent access.

Here's how you can create a thread-local NumberFormat instance:

```
public static final ThreadLocal<NumberFormat>
currencyFormat = ThreadLocal.withInitial(() ->
            NumberFormat.getCurrencyInstance());
```

And then, whenever you need to access the NumberFormat instance within a thread:

```
String amountDue =
             currencyFormat.get().format(total);
```

The ThreadLocal maintains a separate instance of NumberFormat for each thread. The first time get() is called within a thread, it initializes and returns a new NumberFormat instance using the lambda expression provided to withInitial(). Subsequent calls to get() within the same thread return the same instance.

9.7.4 MISCELLANEOUS THREAD PROPERTIES

Threads can be organized into groups, which can be useful for managing and controlling them collectively. API methods are available for managing thread groups, such as interrupting all threads in a group. However, modern practice tends to favor the use of executors for managing groups of tasks due to their higher-level abstractions and ease of use.

Threads can also be assigned priorities, where higher-priority threads are scheduled to run before lower-priority ones. While this feature exists, its reliability depends heavily on the underlying platform and JVM implementation, making it fragile and not generally recommended for use.

Each thread has a state, indicating whether it's new, running, blocked on input/output, waiting, or terminated. In most cases, as an application programmer, you won't need to inquire about thread states.

When a thread terminates due to an uncaught exception, the exception is passed to the thread's uncaught exception handler. By default, the stack trace is dumped to **System.err**, but you can install your own handler if needed.

A **daemon thread** is one that exists solely to serve other threads. These are useful for tasks like sending timer ticks or cleaning up stale cache entries. When only daemon threads remain, the virtual machine exits. To mark a thread as a daemon, you can call thread.setDaemon(true) before starting it.

9.8 ASYNCHRONOUS COMPUTATIONS

9.8.1 LONG-RUNNING TASKS IN USER INTERFACE CALLBACKS

One of the reasons to use threads is to make your programs more responsive. This is particularly important in an application with a user interface. When your program needs to do something time consuming, you cannot do the work in the user-interface thread, or the user interface will be frozen. Instead, fire up another worker thread.

For example, if you want to read a web page when the user clicks a button, don't do this:

```
Button read = new Button("Read");
read.setOnAction(event -> {
   // Bad—action is executed on UI thread
   Scanner in = new Scanner(url.openStream());
   while (in.hasNextLine()) {
      String line = in.nextLine();
      …
   }
});
```

Instead, do the work in a separate thread.

```
read.setOnAction(event -> {
// Good—long-running action in separate thread
   Runnable task = () -> {
      Scanner in = new Scanner(url.openStream());
      while (in.hasNextLine()) {
         String line = in.nextLine();
         …
      }
   }
   new Thread(task).start();
});
```

9.8.2 COMPLETABLE FUTURES

The traditional approach for dealing with nonblocking calls is to **use event handlers**, where the programmer registers a handler for the action that should occur after a task completes.

The **CompletableFuture** class provides an alternative approach. Unlike event handlers, completable futures can be composed.

For example, suppose we want to extract all links from a web page in order to build a web crawler. Let's say we have a method

```
public void CompletableFuture<String>
readPage(URL url)
```

that yields the text of a web page when it becomes available. If the method

```
public static List<URL> getLinks(String page)
```

yields the URLs in an HTML page, you can schedule it to be called when the page is available:

```
CompletableFuture<String> contents =
                                    readPage(url);
CompletableFuture<List<URL>> links =
          contents.thenApply(Parser::getLinks);
```

The thenApply method doesn't block either. It **returns another future**. When the first future has completed, its result is fed to the getLinks method, and the return value of that method becomes the final result.

With completable futures, you just **specify what you want to have done and in which order**. It won't all happen right away, of course, but what is important is that all the code is in one place.

Conceptually, CompletableFuture is a simple API, but there are many variants of methods for composing completable futures.

9.9 PROCESSES

Up to now, you have seen how to execute Java code in separate threads within the same program. Sometimes, you need to execute another program. For this, use the **ProcessBuilder** and **Process classes**. The Process class executes a command in a separate operating system process and lets you interact with its standard input, output, and error streams. The ProcessBuilder class lets you configure a Process object.

9.9.1 BUILDING A PROCESS

Start the building process by **specifying the command that you want to execute.** You can supply a List<String> or simply the strings that make up the command.

```
ProcessBuilder builder = new
ProcessBuilder("gcc", "myapp.c");
```

Caution

The **first string must be an executable command**, **not a shell builtin**.

Each process has a working directory, which is used to resolve relative directory names. By default, **a process has the same working directory as the virtual machine**, which is typically the directory from which you launched the java program.

Note

Each of the methods for configuring a ProcessBuilder returns itself, so that you can chain commands.

Ultimately, you will call:

```
Process p = new ProcessBuilder(command).directo
ry(file)….start();
```

Next, you will want to specify what should happen to the standard input, output, and error streams of the process.

By default, each of them is a pipe that you can access with

```
OutputStream processIn = p.getOutputStream();
InputStream processOut = p.getInputStream();
InputStream processErr = p.getErrorStream();
```

Note that the input stream of the process is an output stream in the JVM! You write to that stream, and whatever you write becomes the input of the process. Conversely, you read what the process writes to the output and error streams. For you, they are input streams.

You can specify that the input, output, and error streams of the new process should be the same as the JVM. If the user runs the JVM in a console, any user input is forwarded to the process, and the process output shows up in the console. Call

```
builder.redirectIO()
```

to make this setting for all three streams. If you only want to inherit some of the streams, pass the value

```
ProcessBuilder.Redirect.INHERIT
```

to the redirectInput, redirectOutput, or redirectError methods.

The files for output and error are created or truncated when the process starts. To append to existing files, use

```
builder.redirectOutput(ProcessBuilder.Redirect.
appendTo(outputFile));
```

It is often useful to merge the output and error streams, so you see the outputs and error messages in the sequence in which the process generates them. Call

```
builder.redirectErrorStream(true)
```

to activate the merging. If you do that, you can no longer call redirectError on the ProcessBuilder or getErrorStream on the Process.

Finally, you may want to modify the environment variables of the process. Here, the builder chain syntax breaks down. You need to get the builder's environment (which is initialized by the environment variables of the process running the JVM), then put or remove entries.

```
Map<String, String> env = builder.environment();
env.put("LANG", "fr_FR");
env.remove("JAVA_HOME");
Process p = builder.start();
```

9.9.2 RUNNING A PROCESS

After you have configured the builder, invoke its **start** method to start the process. If you configured the input, output, and error streams as pipes, you can now write to the input stream and read the output and error streams. For example,

```
Process p = new ProcessBuilder("/bin/ls", "-l")
      .directory(Paths.get("/tmp").toFile())
      .start();
try (Scanner in = new
Scanner(p.getInputStream())) {
    while (in.hasNextLine())
        System.out.println(in.nextLine());
}
```

To wait for the process to finish, call

```
int result = p.waitFor();
```

or, if you don't want to wait indefinitely,

```
long delay = ...;
if (p.waitfor(delay, TimeUnit.SECONDS)) {
    int result = p.exitValue();
...
} else {
    p.destroyForcibly();
}
```

The **first call to waitFor returns the exit value of the process** (by convention, 0 for success or a nonzero error code). The **second call returns true if the process didn't time out**. Then you need to retrieve the exit value by **calling the exitValue method**.

To kill the process, call **destroy** or **destroyForcibly**. The difference between these calls is platform-dependent.

ANNOTATIONS

Annotations are **tags that you insert into your source code so that some tool can process them**. The tools can operate on the source level, or they can process class files into which the compiler has placed annotations.
Annotations do not change the way your programs are compiled.

11.1 USING ANNOTATIONS

Here is an example of a simple annotation:

```
public class CacheTest {
   ...
   @Test public void checkRandomInsertions()
}
```

The annotation **@Test** annotates the checkRandomInsertions method.

In Java, an annotation is used like a modifier (such as public or static). The **name of each annotation is preceded by an @ symbol**.

By itself, the @Test annotation does not do anything. It **needs a tool to be useful**.

11.1.1 ANNOTATION ELEMENTS

Annotations can have **key/value pairs called elements**, such as @Test(timeout=10000)

The names and types of the permissible elements are defined by each annotation. The elements can be processed by the tools that read the annotations.

An annotation element is one of the following:

- A **primitive type value**
- A **String**
- A **Class object**
- An **instance of an enum**
- An **annotation**
- An **array of the preceding** (but not an array of arrays)

For example,

```
@BugReport(showStopper=true,
   assignedTo="Harry",
   testCase=CacheTest.class,
   status=BugReport.Status.CONFIRMED)
```

Caution

An annotation element can **never have the value null**.

Elements can have **default values**. For example, the timeout element of the JUnit @Test annotation has default 0L. Therefore, the annotation @Test is equivalent to @Test(timeout=0L).

If the element name is **value**, and that is the only element you specify, you can omit value=. For example, @SuppressWarnings("unchecked") is the same as @SuppressWarnings(value="unchecked").

If an element value is an **array**, enclose its components in braces: @BugReport(reportedBy={"Harry", "Fred"})
You can **omit the braces if the array has a single component**:

```
@BugReport(reportedBy="Harry")
//Same as {"Harry"}
```

An annotation element can be another annotation:

```
@BugReport(ref=@Reference(id=11235811), ...)
```

Note

Since annotations are evaluated by the compiler, all element values must be compile-time constants.

11.1.2 MULTIPLE AND REPEATED ANNOTATIONS

An item can have multiple annotations:

```
@Test
@BugReport(showStopper=true, reportedBy="Joe")
public void checkRandomInsertions()
```

If the author of an annotation declared it to be repeatable, you can repeat the same annotation multiple times:

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
public void checkRandomInsertions()
```

11.1.3 ANNOTATING DECLARATIONS

Other than in method declaration, there are many other places where annotations can occur. They fall into two categories: **declarations** and **type uses**.

Declaration annotations can appear at the declarations of

- **Classes** (including enum) and **interfaces** (including annotation interfaces)
- **Methods**
- **Constructors**
- **Instance variables** (including enum constants)
- **Local variables** (including those declared in for and try-with-resources statements)
- **Parameter variables** and **catch clause parameters**
- **Type parameters**
- **Packages**

For classes and interfaces, put the annotations before the class or interface keyword:

```
@Entity public class User { ... }
```

For variables, put them before the type:

```
@SuppressWarnings("unchecked") List<User> users=
…;
public User getUser(@Param("id") String userId)
```

A type parameter in a generic class or method can be annotated like this:

```
public class Cache<@Immutable V> { ... }
```

A package is annotated in a file package-info.java that contains only the package statement preceded by annotations.

```
/**
```

```
   Package-level Javadoc
*/
@GPL(version="3")
package com.horstmann.corejava;
import org.gnu.GPL;
```

Note that the import statement for the annotation comes **after** the package declaration.

Note

Annotations for local variables and packages are **discarded when a class is compiled**. Therefore, **they can only be processed at the source level**.

11.1.4 ANNOTATING TYPE USES

A declaration annotation provides some information about the item being declared. For example, in the declaration:

```
public User getUser(@NonNull String userId)
```

it is asserted that the userId parameter is not null.

Note

The @NonNull annotation is a part of the Checker Framework (http://types.cs.washington.edu/checker-framework). With that framework, you can include assertions in your program, such that a parameter is non-null or that a String contains a regular expression. A static analysis tool then checks whether the assertions are valid in a given body of source code.

Now suppose we have a parameter of type List<String>, and we want to express that all of the strings are non-null. That is where type use annotations come in. Place the annotation before the type argument:

```
List<@NonNull String>
```

Type use annotations can appear in the following places:

- With generic type arguments:
  - List<@NonNull String>,
  - Comparator. <@NonNull String> reverseOrder().
- In any position of an array:
  - @NonNull String[][] words (words[i][j] is not null),
  - String @NonNull [][] words (words is not null),
  - String[] @NonNull [] words (words[i] is not null).
- With superclasses and implemented interfaces:
  - class Warning extends @Localized Message.
- With constructor invocations:
  - new @Localized String(...).
- With nested types:
  - Map.@Localized Entry.
- With casts and instanceof checks:
  - (@Localized String) text,
  - if (text instanceof @Localized String). (The annotations are only for use by external tools. They have no effect on the behavior of a cast or an instanceof check.)
- With exception specifications:
  - public String read() throws @Localized IOException.
- With wildcards and type bounds:
  - List<@Localized ? extends Message>,
  - List<? extends @Localized Message>.
- With method and constructor references:
  - @Localized Message::getText.

There are a few type positions that cannot be annotated:

```
@NonNull String.class
// Error—cannot annotate class literal
import java.lang.@NonNull String;
// Error—cannot annotate import
```

You can place annotations **before or after other modifiers** such as private and static. It is customary (but not required) to put type use annotations after other modifiers, and declaration annotations before other modifiers.

11.1.5 MAKING RECEIVERS EXPLICIT

Suppose you want to annotate parameters that are not being mutated by a method.

```
public class Point {
   public boolean equals(@ReadOnly Object
   other) { ... }
}
```

Then a tool that processes this annotation would, upon seeing a call *p.equals(q)* reason that q has not been changed.

But what about p? When the method is called, the receiver variable this is bound to p, but this is never declared, so you cannot annotate it.

Actually, you can declare it, with a rarely used syntax variant, just so that you can add an annotation:

```
public class Point {
   public boolean equals(@ReadOnly Point this,
   @ReadOnly Object other) { ... }
}
```

The first parameter is called the **receiver parameter**. It must be named this. **Its type is the class that is being constructed**.

Note

You can provide a receiver parameter **only for methods, not for constructors**. Conceptually, the this reference in a constructor is not an object of the given type until the constructor has completed. Instead, an a**nnotation placed on the constructor describes a property of the constructed object**.

A different hidden parameter is passed to the **constructor of an inner class**, namely the reference to the enclosing class object. You can make this parameter explicit as well:

```
static class Sequence {
   private int from;
   private int to;
   class Iterator implements
   java.util.Iterator<Integer> {
      private int current;
```

```
      public Iterator(@ReadOnly Sequence
      Sequence.this) {
         this.current = Sequence.this.from;
      }
      ...
   }
   ...
}
```

The parameter must be named just like when you refer to it, EnclosingClass.this, and its type is the enclosing class.

11.2 DEFINING ANNOTATIONS

Each annotation **must be declared by an annotation interface**, with the **@interface syntax**. The methods of the interface correspond to the elements of the annotation. For example, the JUnit Test annotation is defined by the following interface:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {
   long timeout();
   …
}
```

The @interface declaration creates an **actual Java interface**. Tools that process annotations receive objects that implement the annotation interface. When the JUnit test runner tool gets an object that implements Test, it simply invokes the timeout method to retrieve the timeout element of a particular Test annotation.

The element declarations in the annotation interface are **actually method declarations**. The methods of an annotation interface can have no parameters and no throws clauses, and they cannot be generic.

The **Target** and **Retention** annotations are **meta-annotations**. They annotate the Test annotation, indicating the places where the annotation can occur and where it can be accessed.

The **value of the @Target meta-annotation is an array of ElementType objects**, specifying the items to which the annotation can apply. You can specify any number of element types, enclosed in braces. For example,

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface BugReport
```

Note

An annotation without an **@Target restriction** can be used with any declarations but not with type parameters and type uses. (These were the only possible targets in the first Java release that supported annotations.)

The **@Retention meta-annotation specifies where the annotation can be accessed**. There are three choices.

1. **RetentionPolicy.SOURCE:** The annotation is available to source processors, but it is not included in class files.
2. **RetentionPolicy.CLASS:** The annotation is included in class files, but the virtual machine does not load them. This is the default.
3. **RetentionPolicy.RUNTIME:** The annotation is available at runtime and can be accessed through the reflection API.

To specify a default value for an element, add a default clause after the method defining the element. For example:

```
public @interface Test {
   long timeout() default 0L;
   …
}
```

This example shows how to denote a default of an empty array and a default for an annotation.

```
public @interface BugReport {
   String[] reportedBy() default {};
      // Defaults to empty array
   Reference ref() default @Reference(id=0);
      // Default for an annotation
   …
}
```

Caution

Defaults are **not stored with the annotation**; instead, they are dynamically computed. If you change a default and recompile the annotation class, all annotated elements will use the new default, even in class files that have been compiled before the default changed.

You **cannot extend annotation interfaces**, and you **never supply classes that implement annotation interfaces**. Instead, source processing tools and the virtual machine generate proxy classes and objects when needed.

11.3 STANDARD ANNOTATIONS

The Java API defines a number of annotation interfaces in the **java.lang**, **java.lang.annotation**, and **javax.annotation** packages. Four of them are meta-annotations that describe the behavior of annotation interfaces. The others are regular annotations that you use to annotate items in your source code.

11.3.1 ANNOTATIONS FOR COMPILATION

- The **@Deprecated** annotation can be attached to any items whose use is no longer encouraged. The compiler will warn when you use a deprecated item.
- The **@Override** makes the compiler check that the annotated method really overrides a method from the superclass.
- The **@SuppressWarnings** annotation tells the compiler to suppress warnings of a particular type.
- The **@SafeVarargs** annotation asserts that a method does not corrupt its varargs parameter.
- The **@Generated** annotation is intended for use by code generator tools. Any generated source code can be annotated to differentiate it from programmer-provided code.

- The **@FunctionalInterface** annotation is used to annotate conversion targets for lambda expressions.

Of course, you should only add this annotations to interfaces that describe functions. There are other interfaces with a single abstract method (such as AutoCloseable) that are not conceptually functions.

11.3.2 ANNOTATIONS FOR MANAGING RESOURCES

- The **@PostConstruct** and **@PreDestroy** annotations are used in environments that control the lifecycle of objects. Methods tagged with these annotations should be invoked immediately after an object has been constructed or immediately before it is being removed.
- The **@Resource** annotation is intended for resource injection.

11.3.3 META-ANNOTATIONS

- The **@Documented** meta-annotation gives a hint to documentation tools such as Javadoc. Documented annotations should be treated just like other modifiers (such as private or static) for documentation purposes. In contrast, other annotations should not be included in the documentation.

  For example, the @SuppressWarnings annotation is not documented. If a method or field has that annotation, it is an implementation detail that is of no interest to the Javadoc reader. On the other hand, the @FunctionalInterface annotation is documented since it is useful for the programmer to know that the interface is intended to describe a function. Figure 11–1 shows the documentation.

- The **@Inherited** meta-annotation applies only to annotations for classes. When a class has an inherited annotation, then all of its subclasses automatically have the same annotation. This makes it easy to create annotations that work similar to marker interfaces (such as the Serializable interface).

  Suppose you define an inherited annotation @Persistent to indicate that objects of a class can be saved in a database. Then the subclasses of persistent classes are automatically annotated as persistent.

- The **@Repeatable** meta-annotation makes it possible to apply the same annotation multiple times. For historical reasons, the implementor of a repeatable annotation needs to provide a container annotation that holds the repeated annotations in an array.

11.4 PROCESSING ANNOTATIONS AT RUNTIME

Suppose we want to reduce the tedium of implementing toString methods. Of course, one can write a generic toString method using reflection that simply includes all instance variable names and values. But suppose we want to customize that process. We may not want to include all instance variables, or we may want to skip class and variable names. For example, for the Point class we may prefer [5,10] instead of Point[x=5,y=10]. Of course, any number of other enhancements would be plausible, but let's keep it simple. The point is to demonstrate what an annotation processor can do.

Annotate all classes that you want to benefit from this service with the @ToString annotation. In addition, **all instance variables that should be included need to be annotated as well**. The annotation is defined like this:

```
@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface ToString {
   boolean includeName() default true;
}
```

Here are annotated Point and Rectangle classes.

```
@ToString(includeName=false)
public class Point {
   @ToString(includeName=false) private int x;
   @ToString(includeName=false) private int y;
   …
}

@ToString
public class Rectangle {
   @ToString(includeName=false) private Point
   topLeft;
   @ToString private int width;
   @ToString private int height;
   …
}
```

The intent is for a rectangle to be represented as string as Rectangle[[5, 10], width=20,height=30].

At runtime, we cannot modify the implementation of the toString method for a given class. Instead, let us provide a method that can format any object, discovering and using the ToString annotations if they are present.

The key are the methods

- `T getAnnotation(Class<T>)`
- `T getDeclaredAnnotation(Class<T>)`
- `T[] getAnnotationsByType(Class<T>)`
- `T[] getDeclaredAnnotationsByType(Class<T>)`
- `Annotation[] getAnnotations()`
- `Annotation[] getDeclaredAnnotations()`

of the AnnotatedElement interface. The reflection classes Class, Field, Parameter, Method, Constructor, and Package implement that interface.

As with other reflection methods, the methods with Declared in their name yield annotations in the class itself, whereas the others include inherited ones. In the context of annotations, this means that the annotation is @Inherited and applied to a superclass.

If an annotation is not repeatable, call getAnnotation to locate it. For example:

```
Class cl = obj.getClass();
ToString ts = cl.getAnnotation(ToString.class);
if (ts != null && ts.includeName()) …
```

Note that you pass the class object for the annotation (here, ToString.class) and you get back an object of some proxy class that implements the ToString interface. You can then invoke the interface methods to get the values of the annotation elements. If the annotation is not present, the getAnnotation method returns null.

11.5 SOURCE-LEVEL ANNOTATION PROCESSING

Another use for annotation is the automatic processing of source files to produce more source code, configuration files, scripts, or whatever else one might want to generate.

To show you the mechanics, I will repeat the example of generating toString methods. However, this time, let's generate them in Java source. Then the methods will get compiled with the rest of the program, and they will run at full speed instead of using reflection.

11.5.1 ANNOTATION PROCESSORS

Annotation processing is integrated into the Java compiler. During compilation, you can invoke annotation processors by running

```
javac –processor ProcessorClassName1,
ProcessorClassName2, ... sourceFiles
```

The compiler locates the annotations of the source files. Each annotation processor is executed in turn and given the annotations in which it expressed an interest. If an annotation processor creates a new source file, the process is repeated. Once a processing round yields no further source files, all source files are compiled.

Note

An annotation processor can only generate new source files. **It cannot modify an existing source file**.

An annotation processor implements the Processor interface, generally by extending the AbstractProcessor class. You need to specify which annotations your processor supports. In our case:

```
@SupportedAnnotationTypes("com.horstmann.annota
tions.ToString")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
   public class ToStringAnnotationProcessor
   extends AbstractProcessor {
      @Override
      public boolean process(Set<? extends
      TypeElement> annotations,RoundEnvironment
      currentRound) {
            …
   }
}
```

A processor can claim specific annotation types, wildcards such as "com.horstmann.*" (all annotations in the com.horstmann package or any subpackage), or even **"*"** (**all annotations**).

The **process method is called once for each round**, with the set of all annotations that were found in any files during this round, and a **RoundEnvironment reference** that contains information about the current processing round.

11.5.2 THE LANGUAGE MODEL API

You use the language model API for analyzing source-level annotations. Unlike the reflection API, which presents the virtual machine representation of classes and methods, the **language model API lets you analyze a Java program according to the rules of the Java language**.

The compiler produces a tree whose nodes are instances of classes that implement the javax.lang.model.element.Element interface and its subinterfaces, TypeElement, VariableElement, ExecutableElement, and so on. These are the compile-time analogs to the Class, Field/Parameter, Method/Constructor reflection classes.

11.5.3 USING ANNOTATIONS TO GENERATE SOURCE CODE

Let us return to our task of automatically generating toString methods. We can't put these methods into the original classes—annotation processors can only produce new classes, not modify existing ones.

Therefore, we'll add all methods into a utility class ToStrings:

```
public class ToStrings {
    public static String toString(Point obj) {
        Generated code
    }
    public static String toString(Rectangle obj) {
        Generated code
    }
    …
    public static String toString(Object obj) {
        return Objects.toString(obj);
    }
}
```

Since we don't want to use reflection, we annotate accessor methods, not fields:

```
@ToString
public class Rectangle {
    …
    @ToString(includeName=false) public Point getTopLeft
}
    @ToString public int getWidth() { return width; }
    @ToString public int getHeight() { return height; }
}
```

The annotation processor should then generate the following source code:

```
 public static String toString(Rectangle obj) {
     StringBuilder result = new StringBuilder();
     result.append("Rectangle");
     result.append("[");
     result.append(toString(obj.getTopLeft()));
     result.append(",");
     result.append("width=");
     result.append(toString(obj.getWidth()));
     result.append(",");
     result.append("height=");
     result.append(toString(obj.getHeight()));
     result.append("]");
     return result.toString();
 }
```

The **"boilerplate" code** is in gray. Here is an outline of the method that produces the toString method for a class with given TypeElement:

```
private void writeToStringMethod(PrintWriter out, TypeElement te) {
    String className = te.getQualifiedName().toString();
    Print method header and declaration of string builder
    ToString ann = te.getAnnotation(ToString.class);
    if (ann.includeName()) Print code to add class name
    for (Element c : te.getEnclosedElements()) {
        ann = c.getAnnotation(ToString.class);
        if (ann != null) {
            if (ann.includeName()) Print code to add field name
            Print code to append toString(obj.methodName())
        }
    }
    Print code to return string
}
```

And here is an outline of the process method of the annotation processor. It creates a source file for the helper class and writes the class header and one method for each annotated class.

```
public boolean process(Set<? extends TypeElement> annotations,
        RoundEnvironment currentRound) {
    if (annotations.size() == 0) return true;
    try {
        JavaFileObject sourceFile =
processingEnv.getFiler().createSourceFile(
            "com.horstmann.annotations.ToStrings");
        try (PrintWriter out = new PrintWriter(sourceFile.openWriter())) {
            Print code for package and class
            for (Element e :
currentRound.getElementsAnnotatedWith(ToString.class)) {
                if (e instanceof TypeElement) {
                    TypeElement te = (TypeElement) e;
                    writeToStringMethod(out, te);
                }
            }
            Print code for toString(Object)
        } catch (IOException ex) {
            processingEnv.getMessager().printMessage(
                Kind.ERROR, ex.getMessage());
        }
    }
    return true;
}
```