JAVA BasicOnly

note by Masud Rana

- Every Java program contains a main method with his header.
- Programming is the act of designing and implementing computer programs.
- **“Cannot find symbol out”**. This is a compile-time error, compile-time errors are often called syntax errors.
- There are no strict requirements for **pseudocode** because it is read by only human readers, not a computer program.
- A **compile-time error** is a violation of the programming language rules that is detected by the compiler.
- A **run-time error** causes a program to take an action that the programmer did not intend.
- An algorithm for solving a problem is a sequence of steps that is unambiguous, executable, and terminating.

- Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.

●●● String:

String demoString = “GeeksforGeeks”; *String literal*
String demoString = new String (“GeeksforGeeks”); *Creating Object*

- int indexOf (String s, int i);

Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

String s = ”Learn Share Learn”;
int output = s.indexOf("ea",3); *returns 13*

- String trim();

Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.

String word1 = “ Learn Share Learn “;
String word2 = word1.trim(); *returns “Learn Share Learn”*

- String(byte[] byte_arr, int start_index, int length, Charset char_set);

Construct a new string from the bytes array depending on the start_index(Starting location) and length(number of characters from starting location). Uses char_set for decoding.

```
        byte[] b_arr = {71, 101, 101, 107, 115};
         Charset cs = Charset.defaultCharset();
            String s = new String(b_arr, 1, 3, cs);  eek
```

●●● **Here are some important features and methods of the StringBuffer class:**

- StringBuffer objects are **mutable**, meaning that you can change the contents of the buffer without creating a new object.
- The initial capacity of a StringBuffer can be specified when it is created, or it can be set later with the ensureCapacity() method.
- The append() method is used to add characters, strings, or other objects to the end of the buffer.
- The insert() method is used to insert characters, strings, or other objects at a specified position in the buffer.
- The delete() method is used to remove characters from the buffer.
- The reverse() method is used to reverse the order of the characters in the buffer.

```
import java.io.*;

class A {
        public static void main(String args[]){
                StringBuffer sb = new StringBuffer("Hello ");
                  sb.append("Java");    now original string is changed
                    System.out.println(sb);
        }
}
```

Output: **Hello Java**

Compile-time Error

When there is something wrong with **the rules** of the programming language, it detects the issue while compiling the code to a Java virtual machine and the code will throw the error as a result the code will not be executed and won't run . It is also called ***Syntax error.***

Run-time Error
It is often called **logical error**. Mostly the compiler does not recognize the error and runs the program but the output of the program is not as accepted . Sometimes it shows some exceptions in java line 3/0 is an exception .

Class
Class is like a template with a group of variables and Methods for specific objects. It defines the structure and the behavior of an object.

Objects
Objects are the instances of class containing the behavior of the class.

- An object contains state information. An object variable contains an object reference, that is, the location of an object.

Methods
Methods are the set of code that works for a specific field.

- Object is an instance of a class. Class is a blueprint or template from which objects are created.
- Variable value can be over write but can't re declared.

Constructor
Constructors set the initial value for the object every time we create an object.The constructor name must be the same as the Class name.

Unit Testing
Unit testing verifies that a class is working correctly in a isolated situation without the complete program.

Interface
Interface is a class that contains the methods to be followed by the children classes or subclass but it doesn't implement the methods. all the sub classes must implement the method declared in the Interface class. all the methods in the interface are abstract methods .
One single class can inherit from more than one interface.

Polymorphism
In java polymorphism means using same named methods or classes for different uses by using different return types and parameters. something similar to overloading implementation .

●●● Some keyWords

1.Final: The value can be assigned only once and methods can’t be overridden .
2.Protected: variable and the method will only be accessed within the class it is created .
3.Public : Things can be accessed from anywhere within the same package
4.Static: static methods and variables can be used without any object because they are object independent. Common for all the objects.

Abstract classes
Abstract Classes are the super classes in the inheritance where methods and variables are only declared in the super class. Each subclass must Override methods and give the value of the variables. If we implement the abstraction in the above inheritance.

Chapter: (2) Using Objects

An **object** is an entity that you can manipulate by calling one or more of its **methods**. A method consists of a sequence of instructions that can access the internal data of an object.

●●● Assignment:

```
int height;
int width = height; // ERROR—uninitialized variable height
```

The compiler will complain about an "uninitialized variable" when you use a variable that has never been assigned a value.

Calling Methods:

Replacing all occurrences of "issipp" in "Mississippi" with "our".

```
river.replace("issipp", "our");
```

The process of creating a new object is called **construction**.
Argument passed through the object is called **Construction arguments**.

Accessor and Mutator Methods:

A method that accesses an object and returns some information about it, without changing the object, is called an **accessor method.**
In contrast, a method whose purpose is to modify the internal data of an object is called a **mutator method.**

The API is the "application programming interface".
The classes in the standard library are organized into **packages**. A package is a collection of classes with a related purpose.

Object References:

In Java, an object variable does not actually hold an object. It merely holds the memory location of an object.
Objects can be very large. It is more efficient to store only the memory location instead of the entire object.

- Use the double type for floating-point numbers.
- The API (Application Programming Interface) documentation lists the classes and methods of the Java library.

●●● Unit Test Example:

```java
package CashRegister;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

class CashRegisterTest {
   CashRegister cashRegister;
   @BeforeEach
   void setUp() {
       cashRegister = new CashRegister();
   }

   @AfterEach
   void tearDown() {
   }

   @Test
   void recordPurchase() {
       double sum = cashRegister.recordPurchase(2);
       assertEquals(2, sum);
   }

   @Test
   void receivePayment() {
   }

   @Test
   void change() {
   }
}
```

Chapter: (3) IMPLEMENTING CLASSES

●●● Encapsulation:

The process of hiding implementation details while publishing an interface is called encapsulation.

If you hide the implementation details, you can diagnose errors and make changes and improvements to the implementation of a class without disrupting the work of programmers who use the class.

Instance variables are declared within a class but outside of any method, constructor, or block by specifying their access modifier (public, private, protected, or package-private).

- When you implement classes and methods, you should get into the habit of thoroughly commenting their behaviors. In Java there is a very useful standard form for documentation comments. If you use this form in your classes, a program called **javadoc** can automatically generate a neat set of **HTML** pages that describe them.

```
/**
 Behaviour of the method
  @param about parameter
 @return about return
*/
```

- Do not use the void reserved word when you declare a constructor:
  public void BankAccount() *// Error—don't use void!*

- When you implement the method, you provide a parameter variable for each argument. But you don't need to provide a parameter variable for the object on which the method is being invoked. That object is called the **implicit parameter**.
- All other parameter variables are called **explicit parameters.**

- The this reference is used to refer to the implicit parameter of a method.

Chapter: (4) <u>FUNDAMENTAL DATA TYPES</u>

When a value cannot be represented exactly, it is rounded to the
nearest match. The problem arises because computers represent numbers in the binary number system. In the binary number system, there is no exact representation of the fraction.

●●● <u>Constant</u>:

In Java, constants are identified with the reserved word **final**. A variable tagged as **final** can never change after it has been set.

- A **magic number** is a numeric constant that appears in your code without explanation.

Big Numbers:

You have to use methods called add, subtract, and multiply instead arithmetic operator.

```
BigInteger n = new BigInteger("1000000");
BigInteger r = n.multiply(n);
```

The BigDecimal type carries out floating-point computations without roundoff errors.

Arithmetic:

The combination of variables, literals, operators, and/or method calls is called an expression.

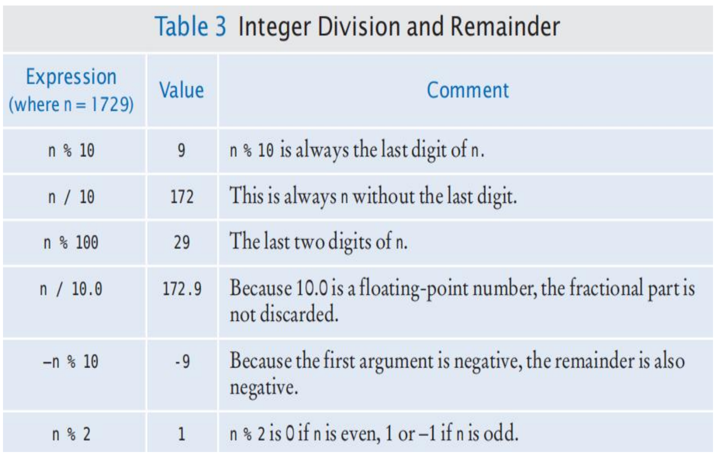


Table 3 Integer Division and Remainder

Expression (where n = 1729)	Value	Comment
`n % 10`	9	`n % 10` is always the last digit of `n`.
`n / 10`	172	This is always `n` without the last digit.
`n % 100`	29	The last two digits of `n`.
`n / 10.0`	172.9	Because 10.0 is a floating-point number, the fractional part is not discarded.
`-n % 10`	-9	Because the first argument is negative, the remainder is also negative.
`n % 2`	1	`n % 2` is 0 if `n` is even, 1 or –1 if `n` is odd.

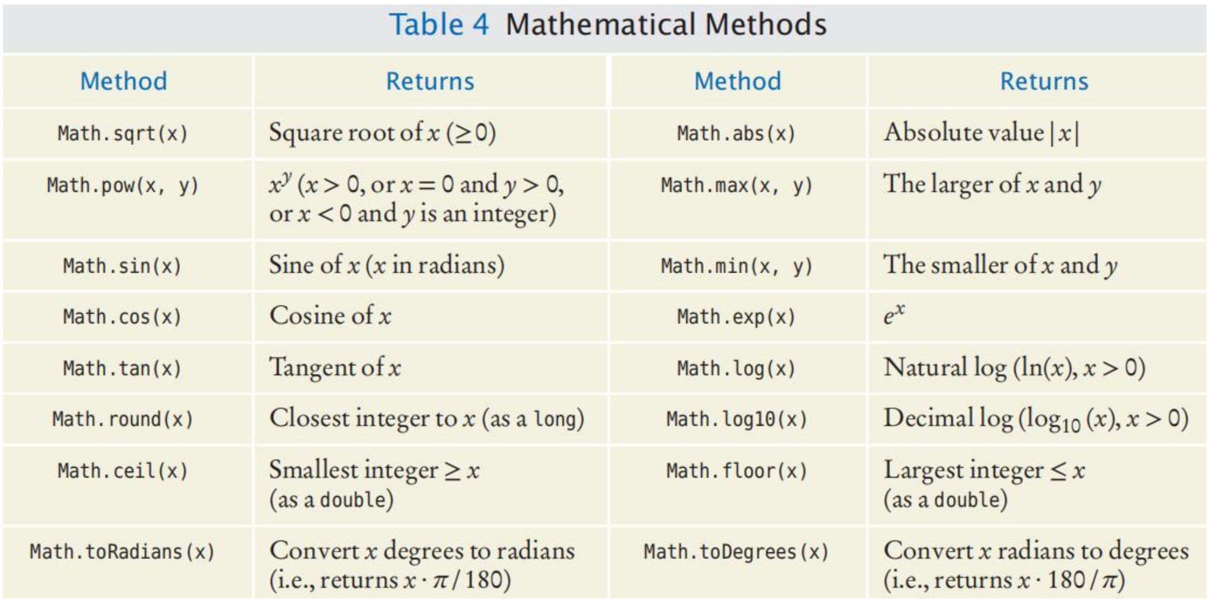


Table 4 Mathematical Methods

Method	Returns	Method	Returns
`Math.sqrt(x)`	Square root of x (≥ 0)	`Math.abs(x)`	Absolute value $\|x\|$
`Math.pow(x, y)`	x^y ($x > 0$, or $x = 0$ and $y > 0$, or $x < 0$ and y is an integer)	`Math.max(x, y)`	The larger of x and y
`Math.sin(x)`	Sine of x (x in radians)	`Math.min(x, y)`	The smaller of x and y
`Math.cos(x)`	Cosine of x	`Math.exp(x)`	e^x
`Math.tan(x)`	Tangent of x	`Math.log(x)`	Natural log ($\ln(x)$, $x > 0$)
`Math.round(x)`	Closest integer to x (as a `long`)	`Math.log10(x)`	Decimal log ($\log_{10}(x)$, $x > 0$)
`Math.ceil(x)`	Smallest integer $\geq x$ (as a `double`)	`Math.floor(x)`	Largest integer $\leq x$ (as a `double`)
`Math.toRadians(x)`	Convert x degrees to radians (i.e., returns $x \cdot \pi / 180$)	`Math.toDegrees(x)`	Convert x radians to degrees (i.e., returns $x \cdot 180 / \pi$)

- We can convert any number type to another any number type like you use a cast (typeName) to convert a value to a different type.

- To Avoiding Negative Remainders, here result of Math.floorMod(m, n) is always positive when n is positive.

The method which can be call without creating object is **static method**. *In this method we can use parameter easily without creating object.*
In contrast, a method that is invoked on an object is called an **instance method**.

●●● Input output:

Create a Scanner object.

```
Scanner in = new Scanner(System.in);
System.out.printf("%10.2f", “%10d”);
```

You can read a string from the console:

```
String name = in.next();   Single word
String name = in.nextLine();      Full sentence
```

Escape Sequences:

To include a quotation mark in a literal string, precede it with a backslash (\).

```
String greeting = "Hello, World!";
String sub = greeting.substring(0, 5); // sub is "Hello"
```

If you omit the end position when calling the substring method, then all characters from the starting position to the end of the string are copied to the variable.

- Java has eight primitive types, including four integer types and two floating-point types.

Chapter: (5) DECISIONS

- Java has a conditional operator of the form
  condition ? value1 : value2

- You have to be careful when comparing floating-point numbers in order to cope with roundoff errors.

- To test whether two strings are equal to each other, you must use the method called equals:
  if (string1.equals(string2)) . . .
  Do not use the == operator to compare strings.
- If two strings are not identical, you still may want to know the relationship between them. The compareTo method compares strings in lexicographic order. This ordering is very similar to the way in which words are sorted in a dictionary.

- You can use the equals method to test whether two rectangles have the same contents.
  box1.equals(box3)

- Sometimes, you need to evaluate a logical condition in one part of a program and use it elsewhere. To store a condition that can be true or false, you use a **Boolean variable**.

!(A && B) is the same as !A || !B
!(A || B) is the same as !A && !B

- Call the hasNextInt or hasNextDouble method to ensure that the next input is a number of Int or Double.

Chapter: (6) LOOPS

```
Random generator = new Random();
int d = 1 + generator.nextInt(6);
```

The call generator.nextInt(6) gives you a random number between 0 and 5 (inclusive). Add 1 to obtain a number between 1 and 6.

Chapter: (7) ARRAYS AND ARRAY LISTS

●●● Declaring and Using Arrays:

Without creating an **Array** with **new** keyword you can't use any array.

```
double[] variable = { 32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65 };
double[] variable = new double[10];
```

To construct an array: new *typeName*[*length*]
To access an element: *arrayReference*[*index*]

```
String[] friends = { "Emily", "Bob", "Cindy" };
```

```
int[] scores = { 10, 9, 7, 4, 5 };
int[] values = scores;
```
Copying array reference

We can pass an whole array as a perimeter.

```
public void addScores(int[] values){
  for (int i = 0; i < values.length; i++) {
   totalScore = totalScore + values[i];}
    }
```

```
int[] scores = { 10, 9, 7, 10 };
addScores(scores);
```

Using Arrays with Methods:

Conversely, a method can return an array.

```
public int[] getScores(){}
```

```
public void addScores(int... values)
```

The int... type indicates that the method can receive any number of int arguments.

The Enhanced for Loop:

Often, you need to visit all elements of an array. The enhanced for loop makes this process particularly easy to program.

```
for (typeName variable : collection) {   The variable contains an element, not an index.
        statements
}
```

```
for (double element : values) {
        element = 0;    ERROR: this assignment does not modify array elements, because it
                        not an index it an element.
}
```

Element Separators:

If you want comma separators, you can use the Arrays.toString method. (You'll need to import java.util.Arrays.)
The expression

Arrays.toString(values)

returns a string describing the contents of the array values in the form

[32, 54, 67.5, 29, 35]

The elements are surrounded by a pair of brackets and separated by commas. This method can be convenient for debugging:

```
System.out.println("values=" + Arrays.toString(values));
```

Removing an Element:

```
for (int i = position + 1; i < currentSize; i++) {
   values[i - 1] = values[i];
 }
currentSize--;
```

Inserting an Element:

```
values[currentSize - 1] =  newElement;
```

It is more work to insert an element at a particular position in the middle of an array. First, move all elements after the insertion location to a higher index. Then insert the new element.

```
if (currentSize < values.length) {
        currentSize++;
for (int i = currentSize - 1; i > pos; i--) {
        values[i] = values[i - 1];
    }
        values[pos] = newElement;
}
```

Array True Copy:

If you want to make a true copy of an array, call the Arrays.copyOf method

```
double[] prices = Arrays.copyOf(values, values.length);
```

The call Arrays.copyOf(values, n) allocates an array of length n, copies the first n elements of values into it, and returns the new array.

●●● Array Lists:

```
import java.util.ArrayList.
```

• Array lists can grow and shrink as needed.
• The Array List class supplies methods for common tasks, such as inserting and removing elements.

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

`friends.add("Cindy");` This will append an element.
`String name = friends.get(index);` Store index element at name.
`friends.set(index, "Harry");` overwrites "Harry" at index.
`friends.remove(index);` Remove element from index position.

Here String is a type parameter. ArrayList is called a generic class. You cannot use primitive types as type parameters.

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

This will print whole array list like *[Emily, Bob, Carolyn]*

Copying Array Lists:

```
ArrayList<String> friends = names;
```

If you want to make a copy of an array list, construct the copy and pass the original list into the constructor:

```
ArrayList<String> newNames = new ArrayList<String>(names);
```

●●● Wrappers and Auto-boxing:

In Java, you cannot directly insert primitive type values—numbers, characters, or boolean values—**into array lists.**
Instead, you must use one of the wrapper classes

Conversion between primitive types and the corresponding wrapper classes is automatic. This process is called **auto-boxing** (even though *auto-wrapping* would have been more consistent).

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Using Array Algorithms with Array Lists:

```
double largest = values[0];
    for (int i = 1; i < values.length; i++) {
          if (values[i] > largest) {
          largest = values[i];
    }
```

Here is the same algorithm, now using an array list:

```
double largest = values.get(0);
    for (int i = 1; i < values.size(); i++) {
          if (values.get(i) > largest) {
    largest = values.get(i);
   }
}
```

Storing Input Values in an Array List:

When you collect an unknown number of inputs, array lists are *much* easier to use than arrays. Simply read inputs and add them to an array list:

```
ArrayList<Double> inputs = new ArrayList<Double>();

        while (in.hasNextDouble()) {
                inputs.add(in.nextDouble());
        }
```

Removing Matches:

```
int i = 0;
    while (i < words.size()) {
        String word = words.get(i);
    if (word.length() < 4) {
    words.remove(i);
        }else {
        i++;
     }
 }
```

Choosing Between Array Lists and Arrays:

• If the size of a collection never changes, use an array.

• If you collect a long sequence of primitive type values and you are concerned about efficiency, use an array.

• Otherwise, use an array list.

The Diamond Syntax:

You can write

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

instead of

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

This shortcut is called the "diamond syntax" because the empty brackets <> look like a diamond shape.

• A test suite is a set of tests for repeated testing.

• Regression testing involves repeating previously run tests to ensure that known failures of prior versions do not appear in new versions of the software.

Chapter: (8) DESIGNING CLASSES

The name for such a class should be a noun that describes the concept.
A class should represent a single concept.

●●● Minimizing Dependencies:

In an object diagram the class names are under lined; in a class diagram the class names are not underlined. In a class diagram, you denote dependency by a **.** dashed line with a -shaped open arrow tip that points to the dependent class.

●●● Guidelines for Designing classes in Java

- **Single Responsibility Principle (SRP):** Each class should have a single responsibility or reason to change.
- **Encapsulation:** Hide the internal details of a class by making fields private and providing public methods for interacting with those fields.Use Meaningful Class Names: Choose descriptive and meaningful names for your classes that reflect their purpose.
- **Keep Classes Small:** Aim for small and focused classes.
- **Use Access Modifiers:** Use access modifiers like public, private, protected, and package-private appropriately to control the visibility and accessibility of class members.
- **Method Naming and Signatures:** Use descriptive method names and well-defined method signatures.
- **Avoid Deep Nesting:** Limit the depth of nesting in your classes.
- **Use Factory and Builder Patterns:** For complex object creation, consider using the Factory or Builder pattern. These patterns provide clear and flexible ways to create objects.
- **Avoid Global State:** Minimize the use of global variables or static state.
- **Documentation and Comments:** Write clear and concise documentation and comment for your classes, methods, and important class members.

For Classes:

Use **Nouns**: Classes represent objects, so use **nouns or noun phrases** that describe what the class represents.

For Methods:

Use **Verbs**: Methods represent actions or behaviors, so **use verbs or verb phrases** for method names.

- To learn more Read mutator and accessoe more deeply.

A **mutator** method changes the state of an object. Conversely, an **accessor** method asks an object to compute a result, without changing the state.

Some classes have been designed to have only accessor methods and no mutator methods at all. Such classes are called **immutable**.

In Java, string objects are **immutable**. Immutable simply means unmodifiable or unchangeable.

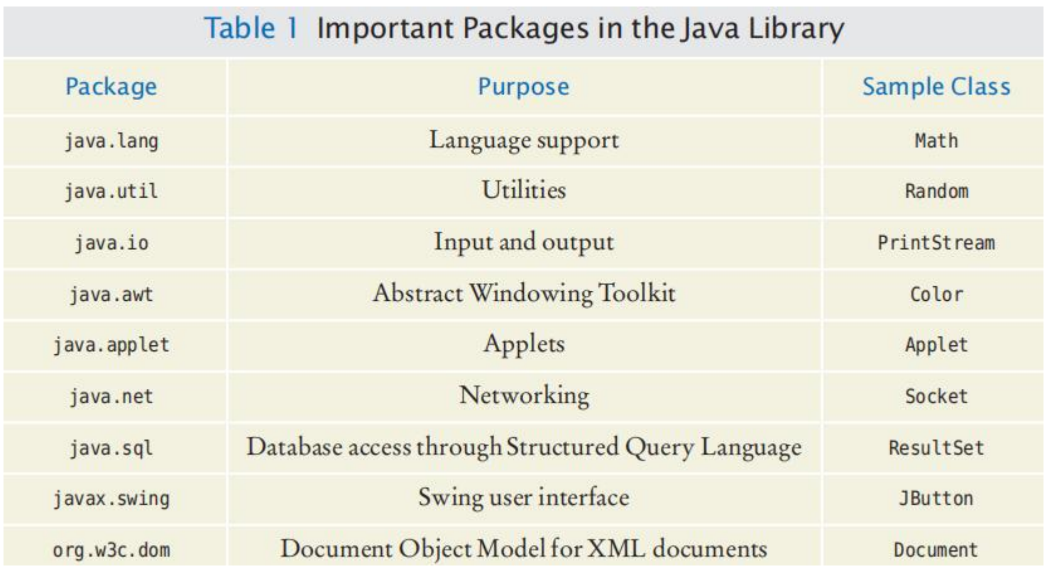
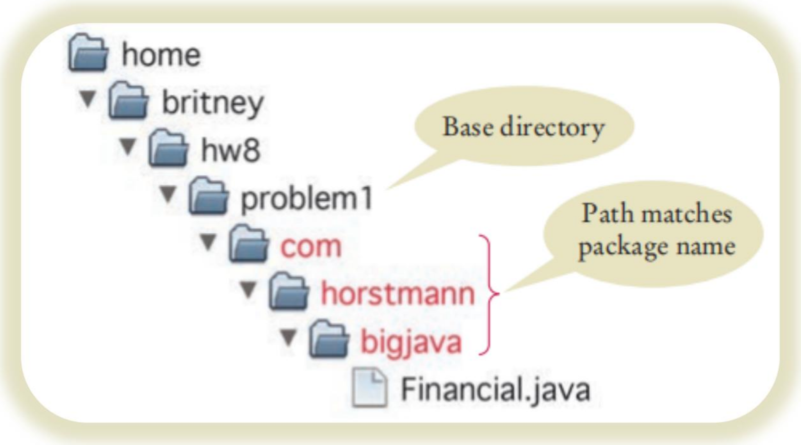


Table 1 Important Packages in the Java Library

Package	Purpose	Sample Class
`java.lang`	Language support	`Math`
`java.util`	Utilities	`Random`
`java.io`	Input and output	`PrintStream`
`java.awt`	Abstract Windowing Toolkit	`Color`
`java.applet`	Applets	`Applet`
`java.net`	Networking	`Socket`
`java.sql`	Database access through Structured Query Language	`ResultSet`
`javax.swing`	Swing user interface	`JButton`
`org.w3c.dom`	Document Object Model for XML documents	`Document`

●●● **Packages:**

```
package packageName;
package com.horstmann.bigjava;
  public class Financial{
          …}
```



Covered review exercise from here

Chapter: (9) INHERITANCE

In object-oriented design, inheritance is a relationship between a more general class (called the super class) and a more specialized class (called the subclass). The subclass inherits data and behavior from the superclass.

```
public class SubclassName extends SuperclassName{
        instance variables
        methods
}
```

●●● Overriding Methods:

The subclass inherits the methods from the superclass. If you are not satisfied with the behavior of an inherited method, you override it by specifying a new implementation in the subclass.

You can call the display method of the superclass, by using the reserved word super:

```
    public void display(){
        super.methodName(parameters);
    }
```

Constructor with Superclass Initializer:

```
    public ClassName(parameterType parameterName, . . .) {
        super(arguments);
    }
```

A class for which you cannot create objects is called an **abstract class**. A class for which you can create objects is sometimes called a **concrete class**.

```
    public abstract class Account {
        public abstract void deductFees();
    }
```

Final Methods and Classes:

prevent other programmers from creating subclasses or from overriding certain methods. In these situations, you use the final reserved word.

```
    public final class String { . . . }
```

That means that nobody can extend the String class.

The equals Method:

The equals method is used to check whether two objects have the same contents:

```
if (stamp1.equals(stamp2))
```

- If you want to represent any object as a string, toString() method comes into existence.
- If you print any object, Java compiler internally invokes the toString() method on the object

The instanceof Operator:

To protect against bad casts, you can use the **instanceof** operator. It tests whether an object belongs to a particular type. Using the instanceof operator, a safe cast can be programmed as follows:

```
if (obj instanceof Question) {
        Question q = (Question) obj;
}
```

- The equals method checks whether two objects have the same contents.

Chapter: (10) INTERFACES

An interface type is similar to a class, but there are several important differences:

• An interface type does not have instance variables.

• Methods in an interface must be abstract (that is, without an implementation) or static, default, or private methods (see Special Topic 10.2).

• All methods in an interface type are automatically public.

• An interface type has no constructor. Interfaces are not classes, and you cannot construct objects of an interface type.

```
public interface InterfaceName {
        method headers
}
```

Implementing an Interface Type:

```
public class ClassName implements InterfaceName, InterfaceName, . . . {
        instance variables
        methods
}
```

Chapter: (11) INPUT/OUTPUT AND EXCEPTION HANDLING

```
File inputFile = new File("input.txt");
Scanner in = new Scanner(inputFile);
```

For file in directory,

```
File inputFile = new File("c:\\homework\\input.dat");
```

Character encoding,

```
Scanner in = new Scanner(file, "UTF-8");
PrintWriter out = new PrintWriter(file, "UTF-8");
```

Reading Words,

```
while (in.hasNext()) {
        String input = in.next();
        System.out.println(input);
}
```

Returns true if ch is a digit ('0' . . . '9'), false otherwise

```
Character.isDigit(ch)
```

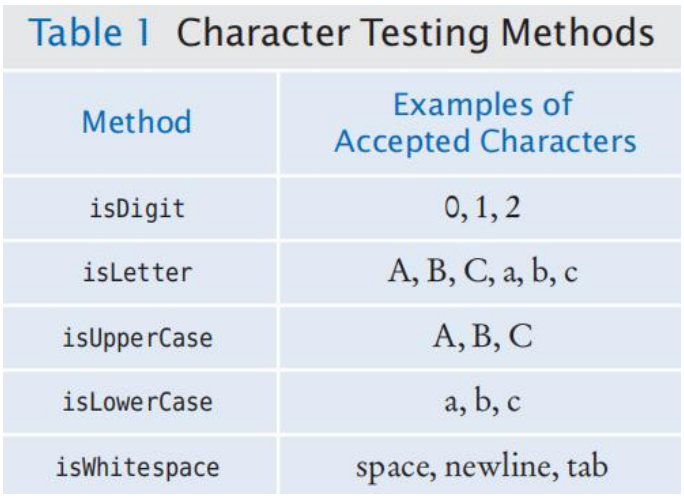


Table 1 Character Testing Methods

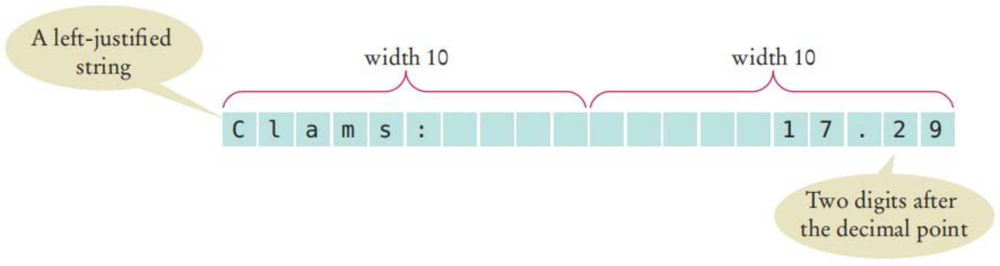
Method	Examples of Accepted Characters
`isDigit`	0, 1, 2
`isLetter`	A, B, C, a, b, c
`isUpperCase`	A, B, C
`isLowerCase`	a, b, c
`isWhitespace`	space, newline, tab

Converting Strings to Numbers

```
int populationValue = Integer.parseInt(population);
```

Formatting Output

```
System.out.printf("%-10s%10.2f", items[i] + ":", prices[i]);
```



```
File inputFile = new File("input.txt");
Scanner in = new Scanner(inputFile);
PrintWriter out = new PrintWriter("output.txt");
```

When you are done processing a file, be sure to close the Scanner or PrintWriter:

```
in.close();
out.close();
```

To achieve read and write exception, we label the main method with a throws declara tion:

public static void main(String[] args) throws **FileNotFoundException**

Double backslash scape single backslash when in a string.

```
File inputFile = new File("c:\\homework\\input.dat");
```

You can use also this:

```
Scanner in = new Scanner(new File("input.txt"));
```

You can read the contents of a web page with this sequence of commands:

```
String address = "http://horstmann.com/index.html";
URL pageLocation = new URL(address);
Scanner in = new Scanner(pageLocation.openStream());
```

Construct a scanner with

```
Scanner in = new Scanner(file, "UTF-8");
```

and a print writer with

```
PrintWriter out = new PrintWriter(file, "UTF-8");
```

The next method of the Scanner class reads the next string.

```
Scanner in = new Scanner(. . .);
in.useDelimiter("[^A-Za-z]+");
```

Here, we set the character pattern that separates words to "any sequence of characters other than letters".

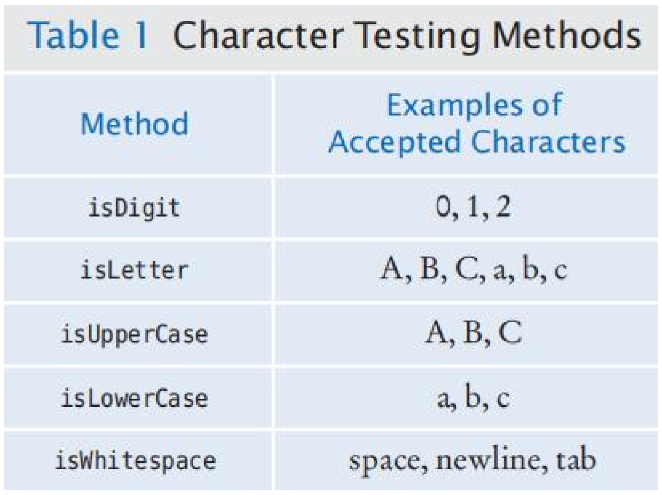


Table 1 Character Testing Methods

Method	Examples of Accepted Characters
`isDigit`	0, 1, 2
`isLetter`	A, B, C, a, b, c
`isUpperCase`	A, B, C
`isLowerCase`	a, b, c
`isWhitespace`	space, newline, tab

Chapter: (18) GENERIC CLASSES

●●● In computer science, generic programming involves the design and implementation of data structures and algorithms that work for multiple types.

Generic Class: A generic class in Java is a class that is parameterized to work with objects of a specific type or types. It uses type parameters enclosed in angle brackets ("< >") to make the class flexible and **compile-time type-safety**, allowing it to work with different data types while maintaining type checking.

> To define a generic class in Java, you use type parameters enclosed in angle brackets ("< >"). These type parameters serve as placeholders for the actual types that will be specified when an instance of the class is created.

Generic Method: A generic method is a method that is parameterized to accept arguments of a specific type or types. It allows methods to work with various data types and ensures type safety by using type parameters. Generic methods can be declared in non-generic classes and interfaces.

Type Safety: Generics provide compile-time type checking, reducing the risk of runtime type-related errors.

Code Reusability: Generic classes can be used with multiple data types, promoting code reusability and reducing redundancy.

Flexibility: Generic classes adapt to different data types, making them versatile and adaptable to changing requirements.

●●●

```
public class Pair<T, S> {
        private T first;
        private S second;
```

Constructs a pair containing two given elements.

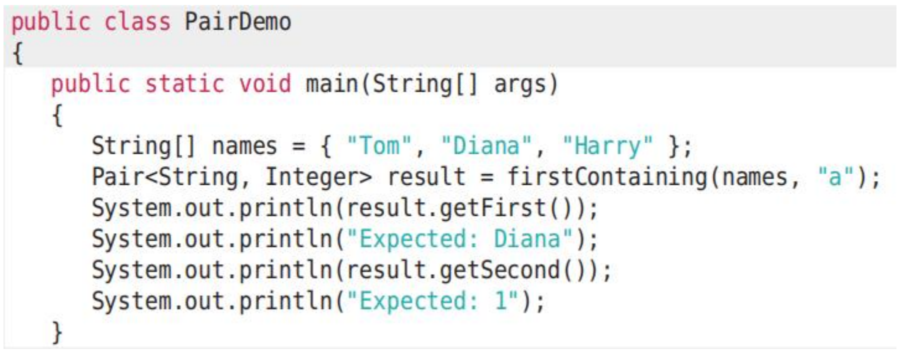
```
public Pair(T firstElement, S secondElement) {
        first = firstElement;
        second = secondElement;
```

Gets the first element of this pair.

```
        public T getFirst() { return first; }
```

Gets the second element of this pair.

```
        public S getSecond() { return second; }
    }
}
```



```
public class PairDemo
{
   public static void main(String[] args)
   {
      String[] names = { "Tom", "Diana", "Harry" };
      Pair<String, Integer> result = firstContaining(names, "a");
      System.out.println(result.getFirst());
      System.out.println("Expected: Diana");
      System.out.println(result.getSecond());
      System.out.println("Expected: 1");
   }
```

●●●

R18.1:

TypeParameter is a generic label used in generic programming to reference an unknown data type, data structure, or class.

R18.2:

A generic class in Java is designed to work with multiple data types using type parameters, ensuring type safety and code reusability. An ordinary class is specific to a single data type and may not provide the same level of flexibility and type safety.

R18.3:

A **generic class** is a class that can work with various data types, specified by type parameters at the class level, making it versatile for different types of data. A **generic method** is a method within any class that can work with different data types, specified by type parameters at the method level, offering flexibility for specific methods to handle various data types.

R18.5:

A commonly used example of a **non-static** generic method in the standard Java library is the **Collections.sort()** method. This method is part of the **java.util.Collections** class and is used to sort elements in a collection.

R18.6:

java.util.Map<K, V>

java.util.AbstractMap.SimpleEntry<K, V>

java.util.AbstractMap.SimpleImmutableEntry<K, V>

java.util.concurrent.atomic.AtomicReferenceFieldUpdater<T, V>

R18.7:

An example of a generic class in the standard Java library that is not a collection class is the **java.util.Optional<T>** class. Optional is used for representing an optional value, which can be either present or absent.

R18.8:

The type parameter **T** in the **binarySearch** method is bound to **Comparable** to ensure that the elements in the array **a** and the **ke**y are comparable with each other. In other words, the **bound T extends Comparable** restricts the type **T** to only those types that implement the Comparable interface.

R18.10:

An **ArrayList<Pair<T, T>>** is a data structure that represents a list of pairs of elements, both of which are of the same type **T**. Each element in the list is a pair, where the first component and the second component of the pair are both of type T. This structure is commonly used to store collections of related values, where each pair represents a related pair of elements.

R18.12:

When you pass an **ArrayList<String>** to a method with an **ArrayList** parameter variable, the code will compile and run without any issues. This is because the ArrayList parameter variable is considered a more general or raw type that can accept any type of objects, including an **ArrayList<String>**.

R18.14:

The code you provided will not compile because of a type incompatibility error. It's not possible to use the **instanceof** operator to check if an **ArrayList<BankAccount>** is an instance of **ArrayList<String**. This is because these two **ArrayList** types are not related by inheritance or interface implementation, and they hold elements of different types.

Chapter: (5) STREAM PROCESSING

Stream API is used to process collections of objects. A stream in Java is a sequence of objects that supports various methods like filtering, mapping,reducing and sorting.

●●● Producing Streams

Stream<String> words = Stream.of("Mary", "had", "a", "little", "lamb");

```
Integer[] digitArray = { 3, 1, 4, 1, 5, 9 };
Stream<Integer> digitStream = Stream.of(digitArray);
```

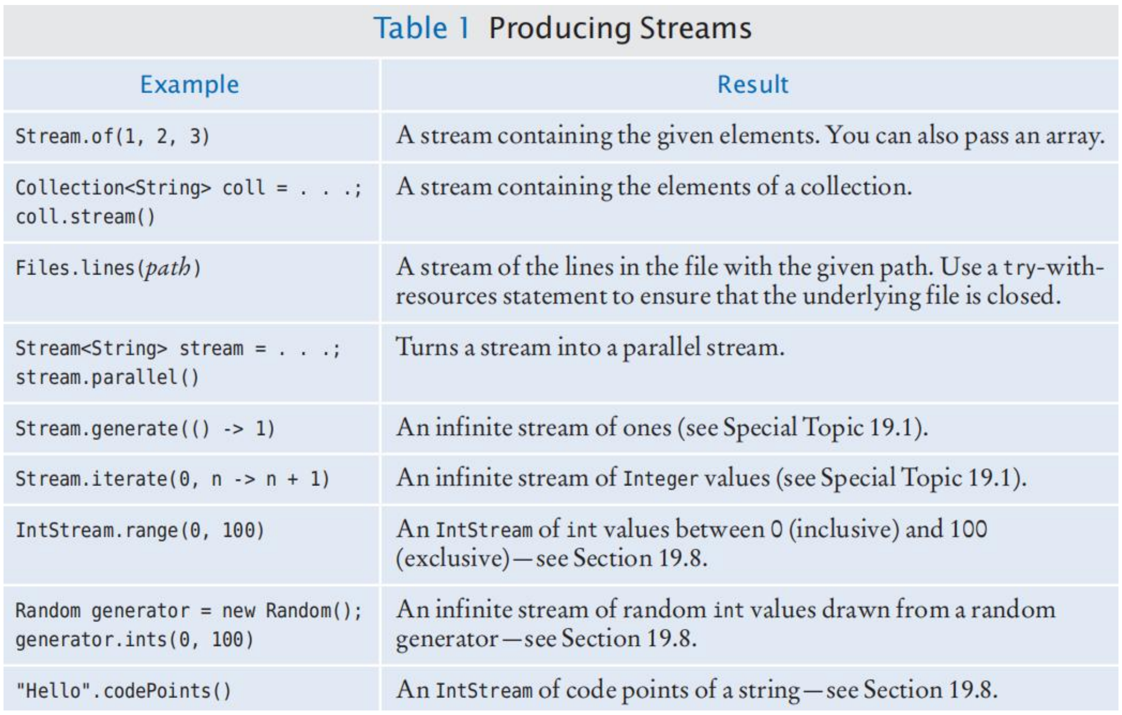


Table 1 Producing Streams

Example	Result
`Stream.of(1, 2, 3)`	A stream containing the given elements. You can also pass an array.
`Collection<String> coll = . . .;` `coll.stream()`	A stream containing the elements of a collection.
`Files.lines(`*path*`)`	A stream of the lines in the file with the given path. Use a `try`-with-resources statement to ensure that the underlying file is closed.
`Stream<String> stream = . . .;` `stream.parallel()`	Turns a stream into a parallel stream.
`Stream.generate(() -> 1)`	An infinite stream of ones (see Special Topic 19.1).
`Stream.iterate(0, n -> n + 1)`	An infinite stream of `Integer` values (see Special Topic 19.1).
`IntStream.range(0, 100)`	An `IntStream` of `int` values between 0 (inclusive) and 100 (exclusive)—see Section 19.8.
`Random generator = new Random();` `generator.ints(0, 100)`	An infinite stream of random `int` values drawn from a random generator—see Section 19.8.
`"Hello".codePoints()`	An `IntStream` of code points of a string—see Section 19.8.

```
        List number = Arrays.asList(2,3,4,5);
        List square = number.stream().map(x->x*x).collect(Collectors.toList());
```

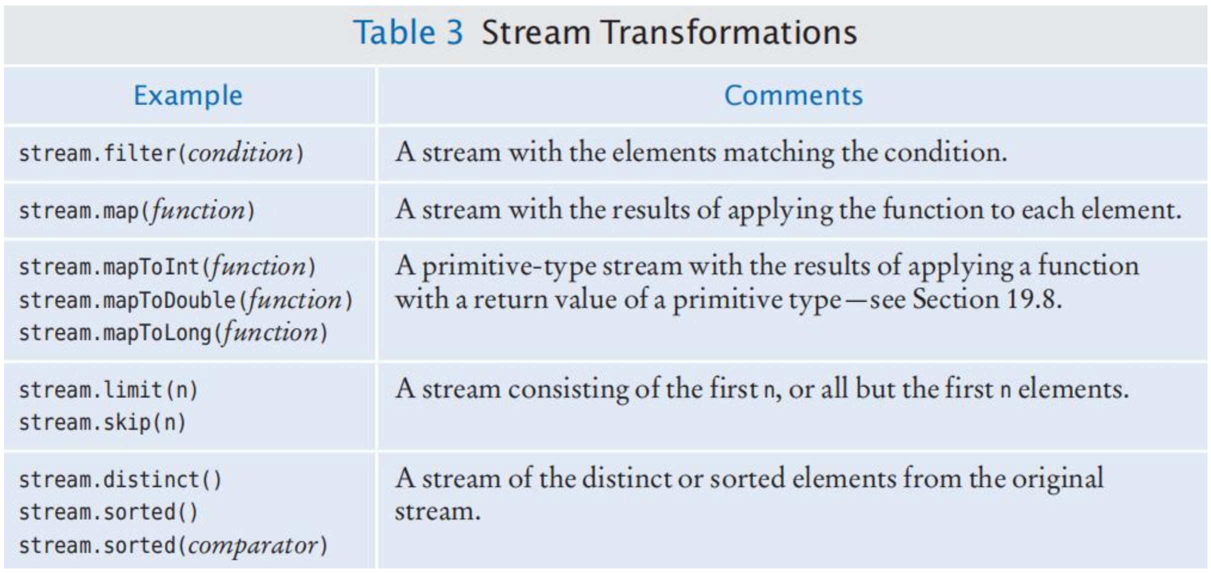


Table 3 Stream Transformations

Example	Comments
`stream.filter(`*condition*`)`	A stream with the elements matching the condition.
`stream.map(`*function*`)`	A stream with the results of applying the function to each element.

`stream.mapToInt(`*function*`)`
`stream.mapToDouble(`*function*`)`
`stream.mapToLong(`*function*`)`
	A primitive-type stream with the results of applying a function with a return value of a primitive type—see Section 19.8.

`stream.limit(n)`
`stream.skip(n)`
	A stream consisting of the first `n`, or all but the first `n` elements.

`stream.distinct()`
`stream.sorted()`
`stream.sorted(`*comparator*`)`
	A stream of the distinct or sorted elements from the original stream.

● ● ●

R19.1:

```
        long countStartsWithA = yourStream.filter(s -> s.startsWith("a")).count();
        long countLongStartsWithA = yourStream.filter(s -> s.length() > 10 &&
        s.startsWith("a")).count();
        boolean atLeast100StartsWithA = yourStream.anyMatch(s ->
        s.startsWith("a"));
```

R19.2:

```
ArrayList<String> words = /* Your ArrayList<String> */;
        ArrayList<String> longWords = new ArrayList<>();
        for (String word : words) {
                if (word.length() > 10) {
                longWords.add(word); if (longWords.size() >= 5) {
                        break;
        } } }
```

R19.3:

The first expression, words.filter(w -> w.length() > 10).limit(100).count(), counts the number of words that have a length greater than 10 characters within the first 100 words of the stream.

The second expression, words.limit(100).filter(w -> w.length() > 10).count(), counts the number of words with a length greater than 10 characters among the entire stream, but it only considers the first 100 words for filtering.

R19.5:

```
List<Integer> integerList = yourStream.collect(Collectors.toList());
Integer[] integerArray = yourStream.toArray(Integer[]::new);
int[] intArray = yourStream.mapToInt(Integer::intValue).toArray();
```

R19.6:

To turn a Stream<Double> into a Stream<String>:

```
Stream<Double> doubleStream = /* Your Stream<Double> */;
Stream<String> stringStream = doubleStream.map(Object::toString);
```

To turn a Stream<String> back into a Stream<Double>:

```
Stream<String> stringStream = /* Your Stream<String> */;
Stream<Double> doubleStream = stringStream.map(Double::valueOf);
```

R19.7:

Using a Method Reference:

```
Function<String, Integer> lambda = String::length;
```

Using a Lambda Expression:

```
Function<String, Integer> lambda = s -> s.length();
```

Using an Anonymous Inner Class (less concise):

```
        Function<String, Integer> lambda = new Function<String, Integer>() {
         @Override
        public Integer apply(String s) {
        return s.length();
    }};
```

R19.9:

Range Closed (Inclusive)

```
	IntStream intStream1 = IntStream.rangeClosed(1, 10);
```

Range (Exclusive)

```
	IntStream intStream2 = IntStream.range(1, 10);
```

Stream of Integers

```
	IntStream intStream3 = Arrays.stream(new int[]{1, 2, 3, 4, 5});
```

Using Stream.mapToInt()

```
Stream<String> stringStream = Stream.of("1", "2", "3", "4", "5");
IntStream intStream4 = stringStream.mapToInt(Integer::parseInt);
```

R19.11:

In Java streams, terminal operations are operations that trigger the processing of the stream and produce a result or a side effect.

For Object Streams (e.g., Stream<T>):

- forEach(Consumer): Performs an action for each element in the stream.
- toArray(): Collects the stream elements into an array.
- reduce(): Aggregates the elements in the stream using a binary operator.
- collect(Collector): Collects elements into a collection using a provided collector.

For Primitive-Type Streams (e.g., IntStream, LongStream, DoubleStream):

- sum(): Calculates the sum of elements.
- average(): Calculates the average of elements.
- min(): Finds the minimum element.
- max(): Finds the maximum element.