Introduction

History of Java

- Java was originally developed by Sun Microsystems starting in 1991
  - James Gosling
  - Patrick Naughton
  - Chris Warth
  - Ed Frank
  - Mike Sheridan
- This language was initially called ***Oak***
- Renamed ***Java*** in 1995

What is Java

- A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language -- **Sun Microsystems**
- **Object-Oriented**
  - No free functions
  - All code belong to some class
  - Classes are in turn arranged in a hierarchy or package structure

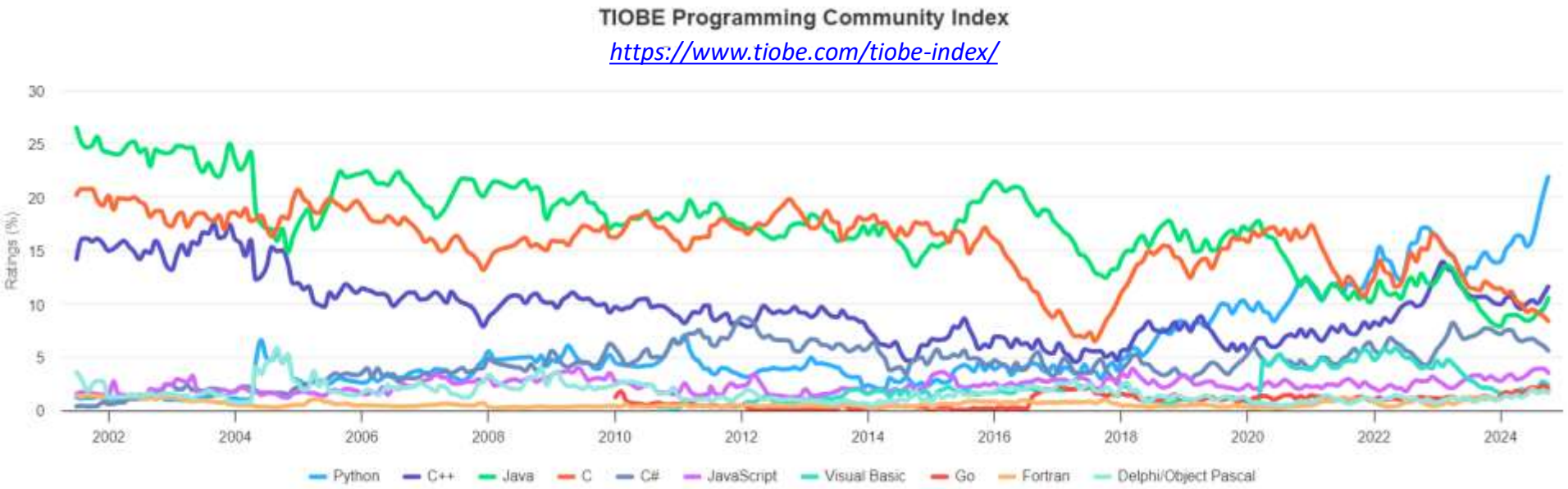
What is Java (2)

- **Distributed**
  - Fully supports IPv4, with structures to support IPv6
  - Includes support for Applets: small programs embedded in HTML documents
- **Interpreted**
  - The program are compiled into Java Virtual Machine (JVM) code called bytecode
  - Each bytecode instruction is translated into machine code at the time of execution

What is Java

- **Robust**
  - Java is simple – no pointers/stack concerns
  - Exception handling – try/catch/finally series allows for simplified error recovery
  - Strongly typed language – many errors caught during compilation

Java in TIOBE Programming Index



Java Editions

- Java 2 Platform, Standard Edition (J2SE)
  - Used for developing desktop-based applications and networking applications
- Java 2 Platform, Enterprise Edition (J2EE)
  - Used for developing large-scale, distributed networking applications and web-based applications
- Java 2 Platform, Micro Edition (J2ME)
  - Used for developing applications for small memory-constrained devices, such as cell phones, pagers and PDAs

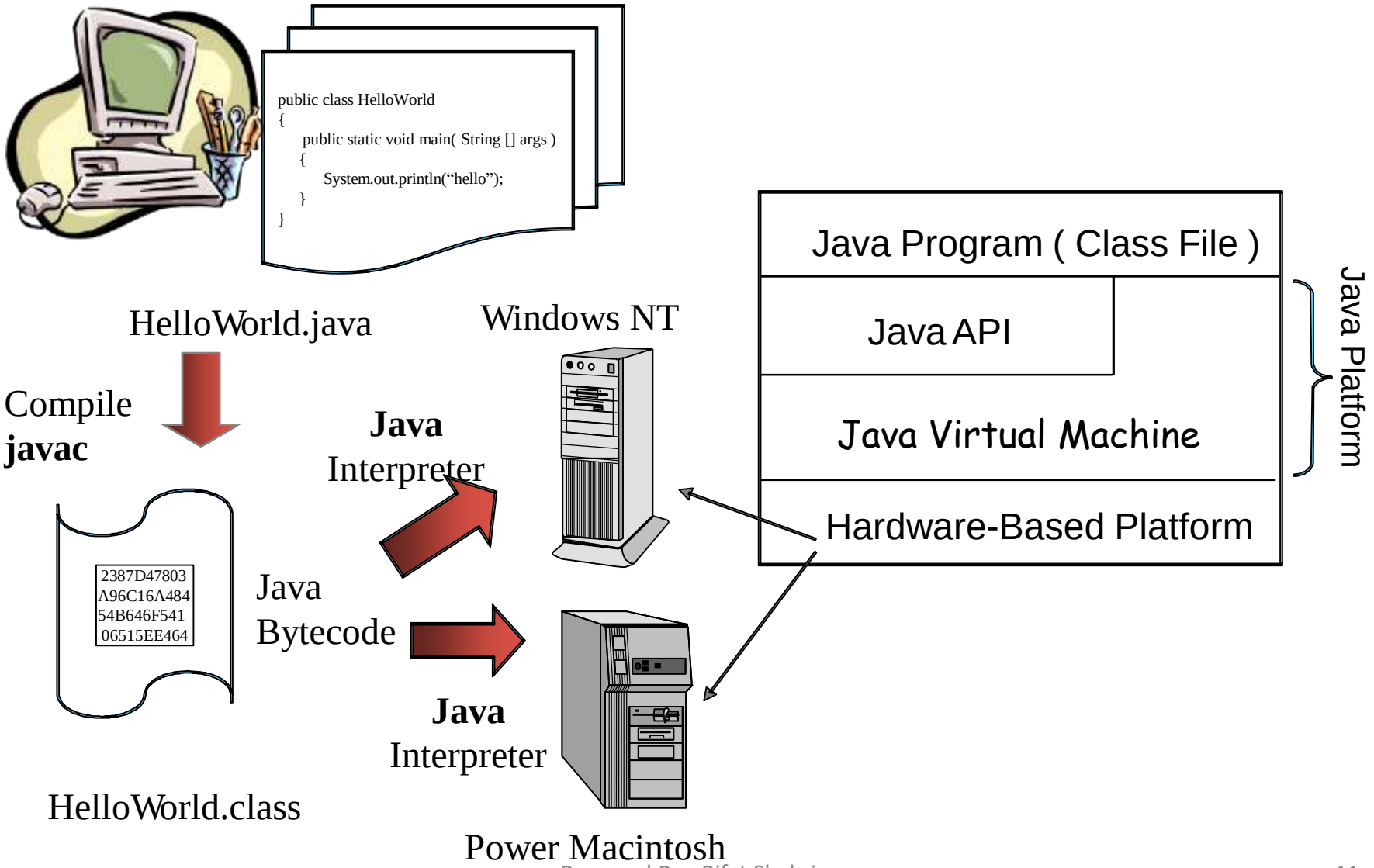
Java SE Versions

Oracle Java SE Support Roadmap*†				
Release	**GA Date**	**Premier Support Until**	**Extended Support Until**	**Sustaining Support**
8 (LTS)**	March 2014	March 2022	December 2030*****	Indefinite
9 - 10 (non-LTS)	September 2017 - March 2018	March 2018 - September 2018	Not Available	Indefinite
11 (LTS)	September 2018	September 2023	January 2032*****	Indefinite
12 - 16 (non-LTS)	March 2019 - March 2021	September 2019 - September 2021	Not Available	Indefinite
17 (LTS)	September 2021	September 2026****	September 2029****	Indefinite
18 - 20 (non-LTS)	March 2022 - March 2023	September 2022 - September 2023	Not Available	Indefinite
21 (LTS)	September 2023	September 2028****	September 2031****	Indefinite
22 (non-LTS)	March 2024	September 2024	Not Available	Indefinite
23 (non-LTS)	September 2024	March 2025	Not Available	Indefinite
24 (non-LTS)***	March 2025	September 2025	Not Available	Indefinite
25 (LTS)***	September 2025	September 2030	September 2033	Indefinite

Installing Java

- Download the JDK
  - From OpenJDK website or Oracle JDK downloads page
- Run the Installer
- Set Environment Variables
  - Add to “PATH” bin directory of JDK installation
  - Set “JAVA_HOME” to path of JDK installation directory
- Verify Installation
  - java –version
  - javac -version

Java platform



The First Java Program

Welcome.java

```
1
2   public class Welcome {
3       public static void main(String[] args) {
4           System.out.println("Hello Java");
5           System.out.printf("I like %s\n", "Java");
6           String strDepartment = "CSE";
7           System.out.print("We study in " + strDepartment + "\n");
8       } // end method main
9   } // end class Welcome - NOTE: no semicolon is required here
10
```

Examining Welcome.java

- A Java source file can contain multiple classes, but only one class can be a public class
- Typically, Java classes are grouped into packages (similar to namespaces in C++)
- A public class is accessible across packages
- The source file name must match the name of the public class defined in the file with the .java extension

Examining Welcome.java

- In Java, there is no provision to declare a class, and then define the member functions outside the class
- Body of every member function of a class (called method in Java) must be written when the method is declared
- Java methods can be written in any order in the source file
- A method defined earlier in the source file can call a method defined later

Examining Welcome.java

- ***public static void main(String[] args)***
  - **main** is the starting point of every Java application
  - **public** is used to make the method accessible by all
  - **static** is used to make main a static method of class Welcome. Static methods can be called without using any object; just using the class name. JVM call main using the **ClassName.methodName** (*Welcome.main*) notation
  - **void** means main does not return anything
  - **String args[ ]** represents an array of String objects that holds the command line arguments passed to the application. *Where is the length of args array?*

Examining Welcome.java

- Think of JVM as a outside Java entity who tries to access the main method of class Welcome
  - main must be declared as a public member of class Welcome
- JVM wants to access main without creating an object of class Welcome
  - main must be declared as static
- JVM wants to pass an array of String objects containing the command line arguments
  - main must take an array of String as parameter

Examining Welcome.java

- ***System.out.println()***
  - Used to print a line of text followed by a new line
  - **System** is a class inside the Java API
  - **out** is a public static member of class System
  - **out** is an object of another class of the Java API
  - **out** represents the standard output (similar to stdout or cout)
  - **println** is a public method of the class of which out is an object

Examining Welcome.java

- **System.out.print()** is similar to **System.out.println()**, but does not print a new line automatically
- **System.out.printf()** is used to print formatted output like printf() in C
- In Java, characters enclosed by double quotes ("") represents a String object, where String is a class of the Java API
- We can use the plus operator (+) to concatenate multiple String objects and create a new String object

Compiling a Java Program

- Open a command prompt window and go to your working directory where the .java file is located
- Execute the following command (path of java bin directory needs to be in PATH environment variable)
  - ***javac Welcome.java***
- If the source code is ok, then javac (the Java compiler) will produce a file called Welcome.class in the current directory

Compiling a Java Program

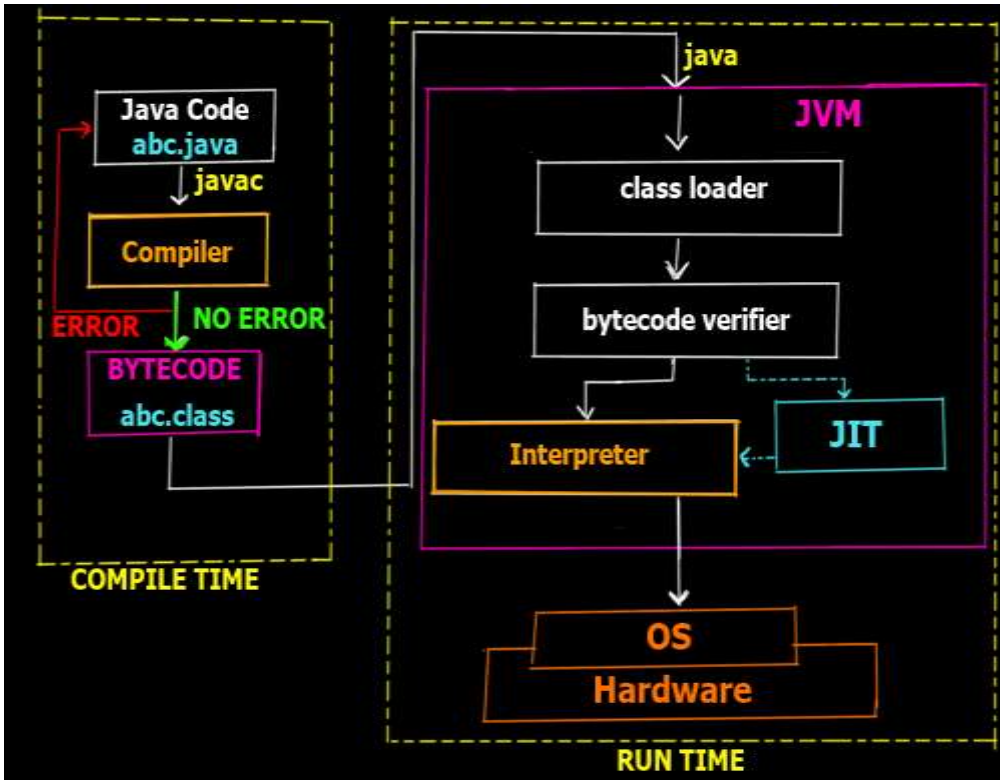
- If the source file contains multiple classes then javac will produce separate .class files for each class
- Every compiled class in Java will have their own .class file
- .class files contain the bytecodes of each class
- So, a .class file in Java contains the bytecodes of a single class only

Executing a Java Program

- After successful compilation execute the following command
  - ***java Welcome***
  - *Note that we have omitted the .class extension here*
- The JVM will look for the class file *Welcome.class* and search for a *public static void main(String args[ ])* method inside the class
- If the JVM finds the above two, it will execute the body of the main method, otherwise it will generate an error and will exit immediately

Java Development Environment

- Edit
  - Create/edit the source code
- Compile
  - Compile the source code
- Load
  - Load the compiled code
- Verify
  - Check against security restrictions
- Execute
  - Execute the compiled



Phase 1: Creating a Program

- Any text editor or Java IDE (Integrated Development Environment) can be used to develop Java programs
- Java source-code file names must end with the ***.java*** extension
- Some popular Java IDEs are
  - IntelliJ
  - VS Code (with appropriate extensions)
  - NetBeans
  - Eclipse

Phase 2: Compiling a Java Program

- ***javac Welcome.java***
  - Searches the file in the current directory
  - Compiles the source file
  - Transforms the Java source code into bytecodes
  - Places the bytecodes in a file named **Welcome.class**

Bytecodes *

- They are not machine language binary code
- They are independent of any particular microprocessor or hardware platform
- They are platform-independent instructions
- Another entity (interpreter) is required to convert the bytecodes into machine codes that the underlying microprocessor understands
- This is the job of the **JVM** (Java Virtual Machine)

JVM (Java Virtual Machine) *

- It is a part of the JDK and the foundation of the Java platform
- It can be installed separately or with JDK
- A virtual machine (VM) is a software application that simulates a computer, but hides the underlying operating system and hardware from the programs that interact with the VM
- It is the JVM that makes Java a portable language

JVM (Java Virtual Machine) *

- The same bytecodes can be executed on any platform containing a compatible JVM
- The JVM is invoked by the java command
  - ***java Welcome***
- It searches the class Welcome in the current directory and executes the main method of class Welcome
- It issues an error if it cannot find the class Welcome or if class Welcome does not contain a method called main with proper signature

Phase 3: Loading a Program *

- One of the components of the JVM is the class loader
- The class loader takes the .class files containing the programs bytecodes and transfers them to RAM
- The class loader also loads any of the .class files provided by Java that our program uses

Phase 4: Bytecode Verification *

- Another component of the JVM is the bytecode verifier
- Its job is to ensure that bytecodes are valid and do not violate Java's security restrictions
- This feature helps to prevent Java programs arriving over the network from damaging our system

Phase 5: Execution

- Now the actual execution of the program begins
- Bytecodes are converted to machine language suitable for the underlying OS and hardware
- Java programs go through two compilation phases
  - Source code -> Bytecodes
  - Bytecodes -> Machine language

Another Java Program

A.java

```
1  public class A {
2      private int a;
3
4      public A()
5      {
6          this.a = 0;
7      }
8
9      public void setA(int a)
10     {
11         this.a = a;
12     }
13
14     public int getA()
15     {
16         return this.a;
17     }
18
19     public static void main(String args[])
20     {
21         A ob;
22         ob=new A();
23         ob.setA(10);
24         System.out.println(ob.getA());
25     }
26 }
27
```

Examining A.java

- The variable of a class type is called a reference
  - *ob* is a reference to A object
- Declaring a class reference is not enough, we have to use new to create an object
- Every Java object has to be instantiated using keyword **new**
- We access a public member of a class using the dot operator (.)
  - Dot (.) is the only member access operator in Java
  - Java does not have **->, &** and *

Primitive (built-in) Data types

- Integers
  - **byte** **8-bit integer (new)**
  - short 16-bit integer
  - int 32-bit signed integer
  - long 64-bit signed integer
- Real Numbers
  - float 32-bit floating-point number
  - double 64-bit floating-point number
- Other types
  - **char** **16-bit, Unicode 2.1 character**
  - boolean true or false, *false is not 0 in Java*

Boolean Type

Boolean.java

```
1  public class Boolean {
2      public static void main(String[] args) {
3          int a = 10;
4          if (a > 0) // if (a) will give compilation error
5          {
6              System.out.println("Inside If");
7          }
8          boolean b = false;
9          if (b)
10         {
11             System.out.println("Inside If");
12         }
13         else
14         {
15             System.out.println("Inside Else");
16         }
17     }
18 }
19
```

Non-primitive Data types

- The non-primitive data types in java are
  - Objects
  - Array
- Non-primitive types are also called reference types

```
public class Box {
    int L, W, H;

    Box(int l, int w, int h)
    {
        L = l;
        W = w;
        H = h;
    }

    public static void main(String[] args)
    {
        Box p; // p is a reference pointing to null
        p = new Box( l: 1, w: 2,  h: 3); // now the actual object is created
    }
}
```

Primitive vs. Non-primitive type

- Primitive types are handled by value – the actual primitive values are stored in variable and passed to methods

  int x = 10;

  public MyPrimitive(int x) { }

- Non-primitive data types (objects and arrays) are handled by reference – the reference is stored in variable and passed to methods

  Box b = new Box(1,2,3);

  public MyNonPrimitive(Box x) { }

Primitive vs. Non-primitive type

- Primitive types are handled by value
  - There is no easy way to swap two primitive integers in Java
  - No method like **void swap(int *x, int *y)**
  - Can only be done using object or array
- But do we actually need a method to swap?
  - **x += (y - (y = x))** does the same in a single statement

Java References

- Java references are used to point to Java objects created by new
- Java objects are **always** passed **by reference** to other functions, ***never by value***
- Java references act as pointers but does not allow pointer arithmetic
- We cannot read the value of a reference and hence cannot find the address of a Java object
- We cannot take the address of a Java reference

Java References

- We can make a Java reference point to a new object
  - By copying one reference to another

    ClassName ref2 = ref1; // Here ref1 is declared earlier
  - By creating a new object and assign it to the reference

    ClassName ref1 = new ClassName();
- We cannot place arbitrary values to a reference except the special value **null** which means that the reference is pointing to nothing

  ClassName ref1 = 100; // compiler error

  ClassName ref2 = null; // no problem

Java References

Box.java

```java
public class Box {
    int L, W, H;

    Box(int l, int w, int h)
    {
        L = l;
        W = w;
        H = h;
    }

    public static void main(String[] args)
    {
        Box b1; // b1 refers to null
        Box b2; // b2 refers to null
        b1 = new Box( l: 8,  w: 5,  h: 7); // b1 refers to new object (8, 5, 7)
        b2 = b1; // b2 refers to b1, so both refers (8, 5, 7)
        b1 = new Box( l: 3,  w: 9,  h: 2); // b1 refers to new object (3, 9, 2)
        b1 = b2; // b1 refers to b2, what happens to object (3, 9, 2)
    }
}
```

Array

Arrays

- A group of variables containing values that all have the same type
- Arrays are fixed-length entities
- In Java, arrays are objects, so they are considered reference types
- But the elements of an array can be either primitive types or reference types

Arrays

- We access the element of an array using the following syntax
  - name[index]
  - “index” must be a nonnegative integer
    - “index” can be int/byte/short/char but not long
- In Java, every array knows its own length
- The length information is maintained in a public final int member variable called **length**

Declaring and Creating Arrays

- int c[ ] = new int [12]
  - Here, “c” is a reference to an integer array
  - “c” is now pointing to an array object holding 12 integers
  - Like other objects arrays are created using “new” and are created in the heap
  - “int c[ ]” represents both the data type and the variable name. Placing number here is a syntax error
  - **int c[12]; // compiler error**

Declaring and Creating Arrays

- int[ ] c = new int [12]
  - Here, the data type is more evident i.e. “int[ ]”
  - But does the same work as
    - int c[ ] = new int [12]
- Is there any difference between the above two approaches?

Declaring and Creating Arrays

- int c[ ], x
  - Here, ‘c’ is a reference to an integer array
  - ‘x’ is just a normal integer variable
- int[ ] c, x;
  - Here, ‘c’ is a reference to an integer array (same as before)
  - But, now ‘x’ is also a reference to an integer array

Arrays

ArrayDemo.java

```
1  public class ArrayDemo {
2      public static void main(String[] args) {
3          int [] a = new int[10];
4          for (int i = 0; i < a.length; i++) {
5              a[i] = i;
6          }
7          for (int i = 0; i < a.length; i++) {
8              System.out.println(a[i]);
9          }
10     }
11 }
12
```

Using an Array Initializer

- We can also use an array initializer to create an array
  - int n[ ] = {10, 20, 30, 40, 50}
- The length of the above array is 5
- n[0] is initialized to 10, n[1] is initialized to 20, and so on
- The compiler automatically performs a “new” operation taking the count information from the list and initializes the elements properly

Arrays of Primitive Types

- When created by “new”, all the elements are initialized with default values
  - byte, short, char, int, long, float and double are initialized to zero
  - boolean is initialized to false
- This happens for both member arrays and local arrays

Arrays of Reference Types

- String [] str = new String[3]
  - Only 3 String references are created
  - Those references are initialized to **null** by default
  - Need to explicitly create and assign actual String objects in the above three positions.
    - str[0] = new String(“Hello”);
    - str[1] = “World”;
    - str[2] = “I” + “ Like” + “ Java”;

Arrays of Reference Types

```
class A {
   private int a;

   public int getA() {
      return a;
   }

   public void setA(int a) {
      this.a = a;
   }
}
```

```
public class ArrayDemo2 {
   public static void main(String[] args) {
      A[] array = new A[10];
      for (int i = 0; i < array.length; i++) {
         array[i] = new A();
         array[i].setA(i);
      }
      for (int i = 0; i < array.length; i++) {
         System.out.println("Object of  A: " + array[i].getA());
      }
   }
}
```

Passing Arrays to Methods

void modifyArray(double d[ ]) {…}

double [] temperature = new double[24];

modifyArray(temperature);

- Changes made to the elements of ‘d’ inside “modifyArray” is visible and reflected in the “temperature” array
- But inside “modifyArray” if we create a new array and assign it to ‘d’ then ‘d’ will point to the newly created array and changing its elements will have no effect on “temperature”

Passing Arrays to Methods

- Changing the elements is visible, but changing the array reference itself is not visible

```
void modifyArray(double d[ ]) {
    d[0] = 1.1; // visible to the caller
}
void modifyArray(double d[ ]) {
    d = new double [10];
    d[0] = 1.1; // not visible to the caller
}
```

Multidimensional Arrays

- Can be termed as array of arrays.
- int b[ ][ ] = new int[3][4];
  - Length of first dimension = 3
    - b.length equals 3
  - Length of second dimension = 4
    - b[0].length equals 4
- int[ ][ ] b = new int[3][4];
  - Here, the data type is more evident i.e. “int[ ][ ]”

Multidimensional Arrays

- int b[ ][ ] = { { 1, 2, 3 }, { 4, 5, 6 } };
  - b.length equals 2
  - b[0].length and b[1].length equals 3
- All these examples represent rectangular two dimensional arrays where every row has same number of columns
- Java also supports jagged array where rows can have different number of columns

Multidimensional Arrays

Example – 1

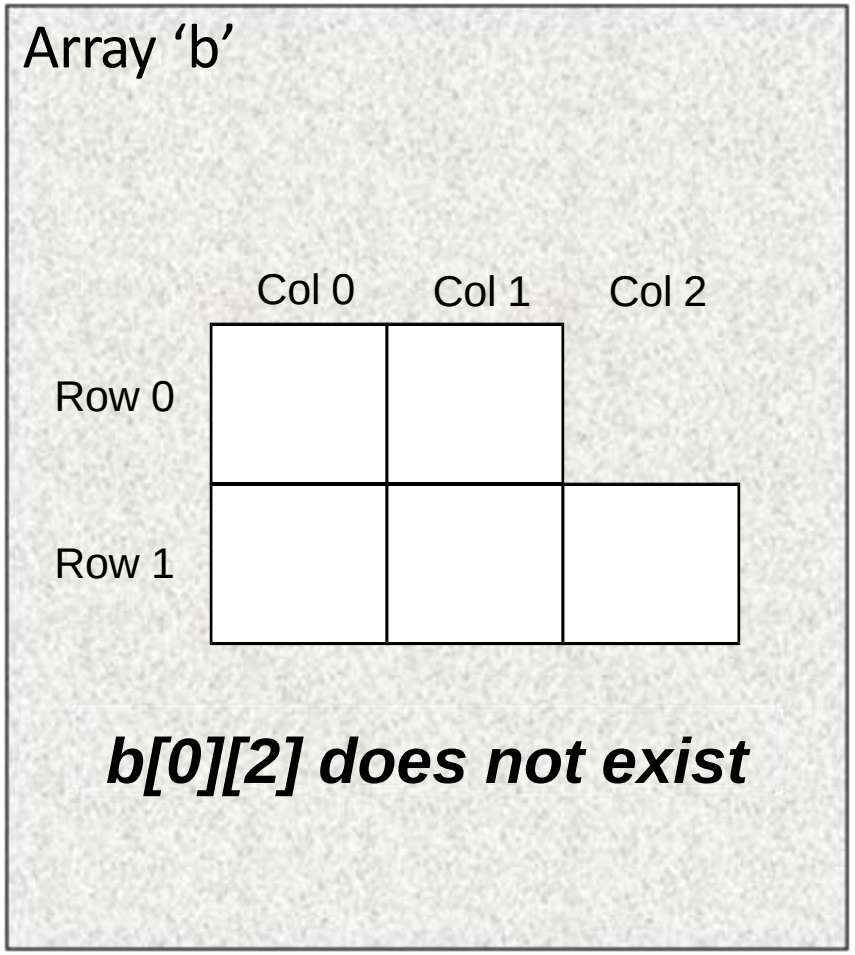
```
int b[ ][ ];
b = new int[2][ ];
b[0] = new int[2];
b[1] = new int[3];
b[0][2] = 7; //will throw an exception
```

Example – 2

```
int b[ ][ ] = { { 1, 2 }, { 3, 4, 5 } };
b[0][2] = 8; //will throw an exception
```

In both cases

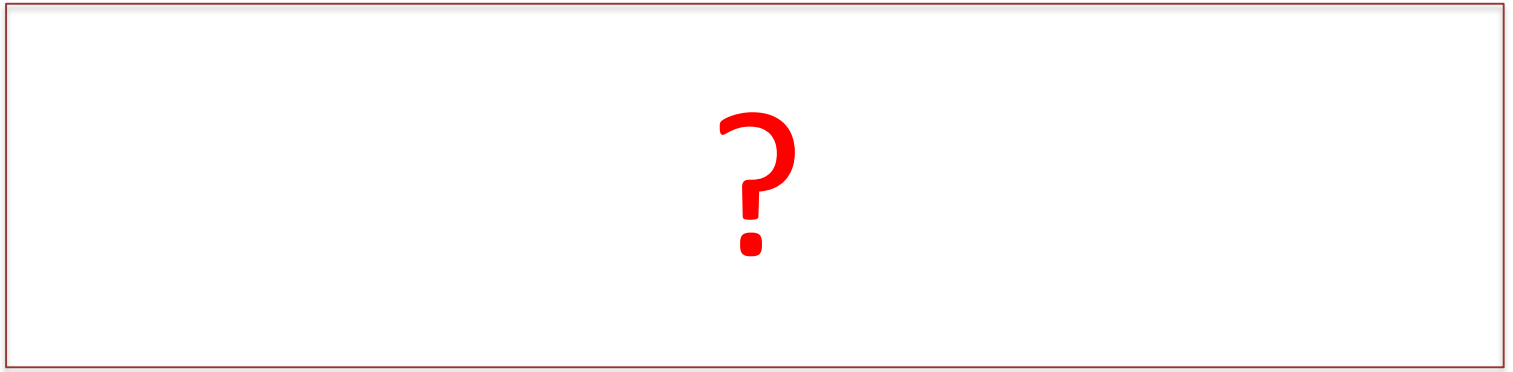
b.length equals 2
b[0].length equals 2
b[1].length equals 3



For-Each loop

```
public class ForEachTest {
    public static void main(String[] args) {
        int numbers[] = {1, 2, 3, 4, 5};
        for (int x : numbers) {
            System.out.print(x + " ");
            x = x * 10; // no effect on numbers
        }
        System.out.println();

        int numbers2[][] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```




```
    }
}
```

Command Line Arguments

Using Command-Line Arguments

- java MyClass arg1 arg2 … argN
  - words after the class name are treated as command-line arguments by Java
  - Java creates a separate String object containing each command-line argument, places them in a String array and supplies that array to main
  - That's why we have to have a String array parameter (String args[ ]) in main
  - We do not need a "argc" type parameter (for parameter counting) as we can easily use "args.length" to determine the number of parameters supplied.

Using Command-Line Arguments

CommandLineTest.java

```
1  public class CommandLineTest {
2      public static void main(String[] args) {
3          System.out.println( args.length );
4
5          for( int i = 0; i < args.length; i++)
6          {
7              System.out.println( args[i] );
8          }
9      }
10 }
11
```

java CommandLineTest Hello 2 You

```
3
Hello
2
You
```

Scanner

Scanner

- It is one of the utility class located in the java.util package
- Using Scanner class, we can take inputs from the keyboard
- Provides methods for scanning
  - int
  - float
  - double
  - line etc.

Scanner

https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/Scanner.html

```java
import java.util.Scanner;

public class ScannerTest {
    public static void main(String[] args) {
        Scanner scn=new Scanner(System.in);
        while(scn.hasNextLine())
        {
            System.out.println(scn.nextLine());
        }
    }
}
```

```java
import java.util.Scanner;

public class ScannerTest {
    public static void main(String[] args) {
        Scanner scn=new Scanner(System.in);
        while(scn.hasNextInt())
        {
            System.out.println(scn.nextInt());
        }
    }
}
```

Issue with Scanner

```
Scanner sc = new Scanner(System.in);

int age = sc.nextInt();
String name = sc.nextLine();

System.out.println(age);
System.out.println(name);
```

```
Scanner sc = new Scanner(System.in);

int age = sc.nextInt();
sc.nextLine(); // consume newline
String name = sc.nextLine();

System.out.println(age);
System.out.println(name);
```

Strings

String related classes

- Java provides four String related classes
- java.lang package
  - ***String*** class: Storing and processing Strings but Strings created using the String class cannot be modified (**immutable**)
  - ***StringBuffer/StringBuilder*** class: Create flexible Strings that can be modified
- java.util package
  - ***StringTokenizer*** class: Can be used to extract tokens from a String

String

- String class provide many constructors and more than 40 methods for examining in individual characters in a sequence
- You can create a String from a String value or from an array of characters.
  - String newString = new String(stringValue);
- The argument stringValue is a sequence of characters enclosed inside double quotes
  - String message = new String (“Welcome”);
  - String message = “Welcome”;

String Constructors

```
3   public class StringConstructorTest {
4       public static void main(String[] args) {
5           char charArray[ ] = { 'b',  'i',  'r',  't',  'h',  ' ',  'd',  'a',  'y'} ;
6           byte byteArray[ ] = { ( byte)  'n',  ( byte)  'e',  ( byte)  'w', ( byte)  ' ',
7                   ( byte)  'y',  ( byte)  'e',  ( byte)  'a',  ( byte)  'r'} ;
8
9           String s = new String("hello"); // hello
10          String s1 = new String() ; //
11          String s2 = new String(s) ; // hello
12          String s3 = new String(charArray); // birth day
13          String s4 = new String(charArray, 6, 3); // day
14          String s5 = new String(byteArray, 4, 4); // year
15          String s6 = new String(byteArray); // new year
16          String s7 = "Wel"  +  "come"; // Welcome
17
18          System.out.println(s);
19          System.out.println(s1);
20          System.out.println(s2);
21          System.out.println(s3);
22          System.out.println(s4);
23          System.out.println(s5);
24          System.out.println(s6);
25          System.out.println(s7);
26      }
27  }
```

String Length

- Returns the length of a String
  - ***length()***
- Example:

```
String s1=“Hello”;
System.out.println(s1.length());
```

Extraction

- Get the character at a specific location in a string
  - ***s1.charAt(1)***
- Get the entire set of characters in a string
  - ***s1.getChars(0, 5, charArray, 0)***

```
public class StringTest {
    public static void main(String[] args) {
        String s = "Hello World";
        char  c1 = s.charAt(6);
        System.out.println(c1);//W
        char [] c2 = new char[5];
        s.getChars(0, 5, c2, 0);
        System.out.println(c2);//Hello
    }
}
```

Extracting Substrings

- substring method enable a new String object to be created by copying part of an existing String object
  - ***substring (int startIndex)*** - copies the characters form the starting index to the end of the String
  - ***substring(int beginIndex, int endIndex)*** - copies the characters from the starting index to one beyond the endIndex

```
public class StringSubstring {
    public static void main(String[] args) {
        String s = "Hello World";
        String s1 = s.substring(6);
        System.out.println(s1); //World
        String s2 = s.substring(2,8);
        System.out.println(s2); //llo Wo
    }
}
```

String Comparisons

- *equals*
  - Compare any two string objects for equality using lexicographical comparison. ***s1.equals("hello")***
- *equalsIgnoreCase*
  - ***s1.equalsIgnoreCase(s2)***
- *compareTo*
  - ***s1.compareTo(s2)***
  - s1 > s2 (positive), s1 < s2 (negative), s1 = s2 (zero)

String Comparisons

```java
public class StringEqualsTest {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = new String( original: "Hello");
        String s3 = "Hello";
        System.out.println("s1 == Hello " + s1.equals("Hello")); // true
        System.out.println("s1 == s2 " + s1.equals(s2)); // true
        System.out.println("s1 == s3 " + s1.equals(s3)); // true
        System.out.println("s2 == s3 " + s2.equals(s3)); // true
        System.out.println(s1 == s2); // false
        System.out.println(s1 == s3); // true
        System.out.println(s2 == s3); // false
    }
}
```

For details have a look at section 1-4 from https://www.baeldung.com/java-string-pool

String Comparisons

- *regionMatches* compares portions of two String objects for equality
  - ***s1.regionMatches (0, s2, 0, 5)***
  - ***s1.regionMatches (true, 0, s2, 0, 5)***
- If the first argument is true, the method ignores the case of the characters being compared
- *startsWith* and endsWith check whether a String starts or ends with a specified String
  - ***s1.startsWith (s2)***
  - ***s1.endsWith (s2)***

String Comparisons

```
public class StringTest {
    public static void main(String[] args) {
        String s = "Hello World";
        String t = “hello ";
        System.out.println(s.regionMatches(true, 0, t, 0, 6));//true
        System.out.println(s.startsWith("Hello")); //true
        System.out.println(s.endsWith("World")); //true
    }
}
```

String Concatenation

- Java provide the *concat* method to concatenate two strings.

```
String s1 = new String (“Happy ”);
String s2 = new String (“Birthday”);
String s3 = s1.concat(s2);
```

s3 will be “Happy Birthday”

String Search

- Find the position of character/String within a String
  - ***int indexOf(char ch)***
  - ***int lastIndexOf(char ch)***

```
public class StringTest {
    public static void main(String[] args) {
        String s = "Hello World";
        System.out.println(s.indexOf('o'));//4
        System.out.println(s.lastIndexOf('o'));//7
    }
}
```

String Conversions

- Generally, the contents of a String cannot be changed once the string is created,
- Java provides conversion methods
- ***toUpperCase()*** and ***toLowerCase()***
  - Converts all the characters in the string to lowercase or uppercase
- ***trim()***
  - Eliminates blank characters from both ends of the string
- ***replace(oldChar, newChar)***
  - Replaces a character in the string with a new character

String to Other Conversions

- The String class provides ***valueOf*** methods for converting a character, an array of characters and numeric values to strings
  - ***valueOf*** method take different argument types

String to Other Conversions

Type	To String	From String
boolean	String.valueOf(boolean)	Boolean.parseBoolean(String)
byte	String.valueOf(int)	Byte.parseByte(String, int base)
short	String.valueOf(int)	Short.parseShort (String, int base)
Int	String.valueOf(int)	Integer.parseInt (String, int base)
long	String.valueOf(long)	Long.parseLong (String, int base)
float	String.valueOf(float)	Float.parseFloat(String)
double	String.valueOf(double)	Double.parseDouble(String)

String Conversion Example

- To convert an int to a String (3 different ways):

  int n = 123;

  String s1 = Integer.toString(n);

  String s2 = String.valueOf(n);

  String s3 = n + "";

- To convert a string to an int:

  String s = "1234";

  int n = Integer.parseInt(s);

String Split

- split() method splits a String against given regular expression and returns a character array

```
public class StringSplitTest {
    public static void main(String[] args) {
        String test = "abc,,def,123";
        String[] out = test.split(",");
        System.out.println(out.length);
        for (int i = 0; i < out.length; i++) {
            System.out.println(out[i]);
        }
    }
}
```

Inheritance

Inheritance

- Same inheritance concept of C++ in Java with some modifications
  - One class inherits the other using ***extends*** keyword
  - The classes involved in inheritance are known as ***superclass*** and ***subclass***
  - ***Multilevel*** inheritance but no ***multiple*** inheritance
  - There is a special way to call the superclass's ***constructor***
  - There is automatic ***dynamic method dispatch***
- Inheritance provides code reusability (code of any class can be used by extending that class)

Simple Inheritance

```
3  class A {
4      int i, j;
5
6      void showij() {
7          System.out.println(i+" "+j);
8      }
9  }
10
11 class B extends A{
12     int k;
13
14     void showk() {
15         System.out.println(k);
16     }
17
18     void sum() {
19         System.out.println(i+j+k);
20     }
21 }
```

```
23 public class SimpleInheritance {
24     public static void main(String[] args) {
25         A superOb = new A();
26         superOb.i = 10;
27         superOb.j = 20;
28         superOb.showij();
29         B subOb = new B();
30         subOb.i = 7;
31         subOb.j = 8;
32         subOb.k = 9;
33         subOb.showij();
34         subOb.showk();
35         subOb.sum();
36     }
37 }
```

Inheritance and Member Access

```java
class M {
    int i;
    private int j;

    void set(int x, int y) {
        i = x;
        j = y;
    }
}

class N extends M {
    int total;

    void sum() {
        total = i + j;
        // Error, j is not accessible here
    }
}
```

```java
public class SimpleInheritance2 {
    public static void main(String[] args) {
        N obj = new N();
        obj.set(10, 20);
        obj.sum();
        System.out.println(obj.total);
    }
}
```

- A class member that has been declared as private will remain private to its class
- It is not accessible by any code outside its class, including subclasses

Practical Example

```
3   class Box {
4       double width, height, depth;
5
6       Box(Box ob) {
7           width = ob.width; height = ob.height; depth = ob.depth;
8       }
9
10      Box(double w, double h, double d) {
11          width = w; height = h; depth = d;
12      }
13
14      Box() { width = height = depth = 1; }
17
18      Box(double len) { width = height = depth = len; }
21
22      double volume() { return width * height * depth; }
25  }
26
27  class BoxWeight extends Box {
28      double weight;
29
30      BoxWeight(double w, double h, double d, double m) {
31          width = w; height = h; depth = d; weight = m;
32      }
33  }
```

Superclass variable reference to Subclass object

```
34
35  public class RealInheritance {
36      public static void main(String[] args) {
37          BoxWeight weightBox = new BoxWeight( w: 3,  h: 5,  d: 7,  m: 8.37);
38          System.out.println(weightBox.weight);
39          Box plainBox = weightBox; // assign BoxWeight reference to Box reference
40          System.out.println(plainBox.volume()); // OK, volume() defined in Box
41          System.out.println(plainBox.weight); // Error, weight not defined in Box
42          Box box = new Box( w: 1,  h: 2,  d: 3); // OK
43          BoxWeight wbox = box; // Error, can't assign Box reference to BoxWeight
44      }
45  }
46
```

Using super to call Superclass Constructors

```
3   class BoxWeightNew extends Box {
4       double weight;
5
6       BoxWeightNew(BoxWeightNew ob) {
7           super(ob);
8           weight = ob.weight;
9       }
10
11      BoxWeightNew(double w, double h, double d, double m) {
12          super(w, h, d);
13          weight = m;
14      }
15
16      BoxWeightNew() {
17          super(); // must be the 1st statement in constructor
18          weight = 1;
19      }
20
21      BoxWeightNew(double len, double m) {
22          super(len);
23          weight = m;
24      }
25
26      void print() {
27          System.out.println("Box(" + width + ", " + height +
28                             ", " + depth + ", " + weight + ")");
29      }
30  }
```

super( ) must always be the first statement executed inside a subclass' constructor

Using super to call Superclass Constructors

```
31
32  public class SuperTest {
33      public static void main(String[] args) {
34          BoxWeightNew box1 = new BoxWeightNew(10, 20, 15, 34.3);
35          BoxWeightNew box2 = new BoxWeightNew(2, 3, 4, 0.076);
36          BoxWeightNew box3 = new BoxWeightNew();
37          BoxWeightNew cube = new BoxWeightNew(3, 2);
38          BoxWeightNew clone = new BoxWeightNew(box1);
39          box1.print();
40          box2.print();
41          box3.print();
42          cube.print();
43          clone.print();
44      }
45
46  }
47
```

Using super to access Superclass hidden members

```
3  class C {
4      int i;
5      void show() {
6      }
7  }
8
9  class D extends C {
10     int i; // this i hides the i in C
11
12     D(int a, int b) {
13         super.i = a; // i in C
14         i = b; // i in D
15     }
16
17     void show() {
18         System.out.println("i in superclass: " + super.i);
19         System.out.println("i in subclass: " + i);
20         super.show();
21     }
22 }
23
24 public class UseSuper {
25     public static void main(String[] args) {
26         D subOb = new D( a: 1,  b: 2);
27         subOb.show();
28     }
29 }
```

Multilevel Inheritance

```
3  class X {
4      int a;
5      X() {
6          System.out.println("Inside X's constructor");
7      }
8  }
9
10 class Y extends X {
11     int b;
12     Y() {
13         System.out.println("Inside Y's constructor");
14     }
15 }
16
17 class Z extends Y {
18     int c;
19     Z() {
20         System.out.println("Inside Z's constructor");
21     }
22 }
23
24 public class MultilevelInheritance {
25     public static void main(String[] args) {
26         Z z = new Z();
27         z.a = 10;
28         z.b = 20;
29         z.c = 30;
30     }
31 }
```

Inside X's constructor
Inside Y's constructor
Inside Z's constructor

Method Overriding

```java
class Base {
    int a;
    Base(int a) {
        this.a = a;
    }
    void show() {
        System.out.println(a);
    }
}

class Child extends Base {
    int b;

    Child(int a, int b) {
        super(a);
        this.b = b;
    }

    // the following method overrides Base class's show()
    @Override // this is an annotation (optional but recommended)
    void show() {
        System.out.println(a + ", " + b);
    }
}
```

```java
public class MethodOverride {
    public static void main(String[] args) {
        Child o = new Child( a: 10,  b: 20);
        o.show();
        Base b = o;
        b.show(); // will call show of Override
    }
}
```

Dynamic Method Dispatch

```
3  class P {
4      void call() {
5          System.out.println("Inside P's call method");
6      }
7  }
8  class Q extends P {
9      void call() {
10         System.out.println("Inside Q's call method");
11     }
12 }
13 class R extends Q {
14     void call() {
15         System.out.println("Inside R's call method");
16     }
17 }
18
19 public class DynamicDispatchTest {
20     public static void main(String[] args) {
21         P p = new P(); // object of type P
22         Q q = new Q(); // object of type Q
23         R r = new R(); // object of type R
24         P  x;          // reference of type P
25         x = p;         // x refers to a P object
26         x.call();      // invoke P's call
27         x = q;         // x refers to a Q object
28         x.call();      // invoke Q's call
29         x = r;         // x refers to a R object
30         x.call();      // invoke R's call
31     }
32 }
```

```
class Figure {
    double d1, d2;
    Figure(double a, double b) { this.d1 = a; this.d2 = b;}
    }
    public double area() {
        System.out.println(
        return 0;
    }
}
```

```
class Rectangle extends Fig
    Rectangle(double a, dou
    @Override
    public double area() {
        System.out.println(
        return d1*d2;
    }
}
```

```
class Triangle extends Figu
    Triangle(double a, doub
    @Override
    public double area() {
        System.out.println("Inside area for Triangle");
        return (d1*d2)/2;
    }
}
```

```
public class FindAreas {
    public static void main(String[] args) {
        Figure f = new Figure(10, 10);
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);

        Figure ref;

        ref = f;
        System.out.println("Area: " + ref.area());

        ref = r;
        System.out.println("Area: " + ref.area());

        ref = t;
        System.out.println("Area: " + ref.area());
    }
}
```

Use of instanceof

instanceof is used to check whether an object belongs to a particular class or interface at runtime.

```
String s = "Hello";
System.out.println(s instanceof String);   // true
System.out.println(s instanceof Object);   // true
```

```
class Animal {}
class Dog extends Animal {}
Animal a = new Dog();
System.out.println(a instanceof Animal);  // true
System.out.println(a instanceof Dog);     // true
```

Abstract Class

- ***abstract class A***
- contains abstract method ***abstract method f()***
- No instance can be created of an abstract class
- The subclass must implement the abstract method
- Otherwise the subclass will be a abstract class too

Abstract Class

```java
abstract class S {
    // abstract method
    abstract void call();
    // concrete methods are still allowed in abstract classes
    void call2() {
        System.out.println("This is a concrete method");
    }
}

class T extends S {
    void call() {
        System.out.println("T's implementation of call");
    }
}

class AbstractDemo {
    public static void main(String args[]) {
        //S s = new S(); // S is abstract; cannot be instantiated
        T t = new T();
        t.call();
        t.call2();
    }
}
```

```
abstract class Figure2 {
    double d1, d2;
    Figure2 (double a, double b) {
        this.d1 = a;
        this.d2 = b;
    }
    abstract double area();
}
```

```
class Rectangle2 extends Fi
    Rectangle2(double a, do
    @Override
    public double area() {
        System.out.println(
        return d1*d2;
    }
}
```

```
class Triangle2 extends Fig
    Triangle2(double a, dou
    @Override
    public double area() {
        System.out.println("Inside area for Triangle");
        return (d1*d2)/2;
    }
}
```

```
public class FindAreas2 {
    public static void main(String[] args) {
        Rectangle2 r = new Rectangle2(9, 5);
        Triangle2 t = new Triangle2(10, 8);

        Figure2 ref;

        ref = r;
        System.out.println("Area: " + ref.area());

        ref = t;
        System.out.println("Area: " + ref.area());
    }
}
```

Anonymous Subclass

```
3  abstract class S {
4      // abstract method
5      abstract void call();
6      // concrete methods are still allowed in abstract classes
7      void call2() {
8          System.out.println("This is a concrete method");
9      }
10 }
11
12 class AbstractDemo {
13     public static void main(String args[]) {
14         //S s = new S(); // S is abstract; cannot be instantiated
15         S s = new S() {
16             void call() {
17                 System.out.println("Call method of an abstract class");
18             }
19         };
20         s.call();
21     }
22 }
```

Using final with Inheritance

To prevent overriding

```
class A {
    final void f() {
        System.out.println("This is a final method.");
    }
}

class B extends A {
    void f() { // Error! Can't override.
        System.out.println("Illegal!");
    }
}
```

To prevent inheritance

```
final class A {
    //...
}

// The following class is illegal.
class B extends A { // Error! Can't subclass A
    //...
}
```

Interface

Interface

- We can call it a pure abstract class having no concrete methods
  - All methods declared in an interface are implicitly **public** and **abstract**
  - All variables declared in an interface are implicitly **public**, **static** and **final**
- *An interface can't have instance variables, so can't maintain state information unlike class*
- A class can only extend from a **single class**, but a class can implement **multiple interfaces**

Implementing Interface

- When you implement an interface method, it must be declared as public
- By implementing an interface, a class signs a contract with the compiler that it will definitely provide implementation of all the methods
  - If it fails to do so, the class will be considered as abstract
  - Then it must be declared as abstract and no object of that class can be created
- An abstract class specifies **what an object is** and an interface specifies **what the object can do**

Simple Interface

```
1  interface Callback {
2      void call(int param);
3  }
4
5  class Client implements Callback {
6      public void call(int p) {
7          System.out.println("call method called with " + p);
8      }
9      public void f() {
10         System.out.println("simple method, not related with Callback");
11     }
12 }
13 public class InterfaceTest {
14     public static void main(String[] args) {
15         // Error, Callback is abstract, can't be instantiated
16         // Callback c = new Callback();
17         // Can't instantiate an interface directly
18         Client client = new Client();
19         client.call( p: 42);
20         client.f();
21         // Accessing implementations through Interface reference
22         Callback cb = new Client();
23         cb.call( param: 84);
24         // cb.f(); Error, no such method in Callback
25     }
26 }
```

Simple Interface

```java
interface Callback {
    void call(int param);
}

public class InterfaceTest {
    public static void main(String[] args) {
        // Anonymous class that implements Callback, introduced in Java 8
        Callback callback = new Callback() {
            @Override
            public void call(int param) {
                System.out.println("call method called with " + param);
            }
        };
        callback.call( param: 10);
    }
}
```

Applying Interfaces

```java
interface MyInterface {
    void print(String msg);
}

class MyClass1 implements MyInterface {
    public void print(String msg) {
        System.out.println(msg + ":" + msg.length());
    }
}

class MyClass2 implements MyInterface {
    public void print(String msg) {
        System.out.println(msg.length() + ":" + msg);
    }
}
public class InterfaceApplyTest {
    public static void main(String[] args) {
        MyClass1 mc1 = new MyClass1();
        MyClass2 mc2 = new MyClass2();
        MyInterface mi; // create an interface reference variable
        mi = mc1;
        mi.print("Hello World");
        mi = mc2;
        mi.print("Hello World");
    }
}
```

Variables in Interfaces

```
1  import java.util.Random;
2
3  interface SharedConstants {
4      int NO = 1;
5      int YES = 2;
6  }
7
8  class Question implements SharedConstants {
9      Random rand = new Random();
10     int ask() {
11         int prob = (int) (100 * rand.nextDouble());
12         if (prob < 50) return NO;
13         else return YES;
14     }
15 }
16 public class InterfaceVariableTest {
17     public static void main(String[] args) {
18         Question q = new Question();
19         for (int i = 0; i < 10; i++) {
20             System.out.println(q.ask());
21         }
22     }
23 }
24
```

Extending Interfaces

```java
interface I1 {
    void f1();
}
interface I2 {
    void f2();
}
interface I3 extends I1, I2 {
    void f3();
}
class MyClass implements I3 {
    public void f1() { System.out.println("Implement f1"); }
    public void f2() { System.out.println("Implement f2"); }
    public void f3() { System.out.println("Implement f3"); }
}

public class InterfaceExtendsTest {
    public static void main(String[] args) {
        MyClass m = new MyClass();
        m.f1();
        m.f2();
        m.f3();
    }
}
```

Default Interface Methods

- Prior to Java 8, an interface could not define any implementation whatsoever
- The release of Java 8 has changed this by adding a new capability to interface called the *default method*
  - A default method lets you define a default implementation for an interface method
  - Its primary motivation was to provide a means by which interfaces could be expanded without breaking existing code

Default Interface Methods

```
1  interface MyIF {
2      // This is a "normal" interface method declaration.
3      int getNumber();
4      // This is a default method. Notice that it provides
5      // a default implementation.
6      default String getString() { return "Default String"; }
9  }
10
11 class MyIFImp implements MyIF {
12     // Only getNumber() defined by MyIF needs to be implemented.
13     // getString() can be allowed to default.
14     public int getNumber() { return 100; }
17 }
18
19 public class InterfaceDefaultMethodTest {
20     public static void main(String[] args) {
21         MyIFImp m = new MyIFImp();
22         System.out.println(m.getNumber());
23         System.out.println(m.getString());
24     }
25 }
26
```

Multiple Inheritance Issues

```java
interface Alpha {
    default void reset() {
        System.out.println("Alpha's reset");
    }
}

interface Beta {
    default void reset() {
        System.out.println("Beta's reset");
    }
}

class TestClass implements Alpha, Beta {
    public void reset() {
        System.out.println("TestClass's reset");
    }
}
```

```java
interface Alpha {
    default void reset() {
        System.out.println("Alpha's reset");
    }
}

interface Beta extends Alpha {
    default void reset() {
        System.out.println("Beta's reset");
        // Alpha.super.reset();
    }
}

class TestClass implements Beta {

}
```

Static Methods in Interface

```java
interface MyIFStatic {
    int getNumber();

    default String getString() {
        return "Default String";
    }

    // This is a static interface method (introduced in Java 8)
    // not inherited by either an implementing class or a subinterface.
    static int getDefaultNumber() {
        return 0;
    }
}

public class InterfaceStaticMethodTest {
    public static void main(String[] args) {
        System.out.println(MyIFStatic.getDefaultNumber());
    }
}
```

Private Methods in Interface

```java
interface MyIFPrivate {
    default String f1() {
        login();
        return "Hello";
    }
    default String f2() {
        login();
        return "World";
    }
    // This is a private interface method (introduced in Java 9)
    // can be called only by a default method or another private method of the same interface
    private void login() {
        System.out.println("login");
    }
}
class MyIFPrivateImp implements MyIFPrivate {
}
public class InterfacePrivateMethodTest {
    public static void main(String[] args) {
        MyIFPrivate ifp = new MyIFPrivateImp();
        System.out.println(ifp.f1());
        System.out.println(ifp.f2());
    }
}
```

Exception

Exception Handling

- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error
  - That method may choose to handle the exception or pass it on (caught and processed at some point)
- Generated by the Java runtime or by your code
  - Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment
  - Manually generated exceptions are typically used to report some error condition to the caller of a method

Exception Handling

- Java exception handling is managed via five keywords
  - Program statements that you want to monitor for exceptions are contained within a ***try*** block
  - If an exception occurs within the try block, it is thrown
  - Your code can catch this exception (using ***catch***)
  - To manually throw an exception, use the keyword ***throw***
  - Any exception that is thrown out of a method must be specified as such by a ***throws*** clause
  - Any code that absolutely must be executed after a try block completes is put in a ***finally*** block

Exception Classes Hierarchy

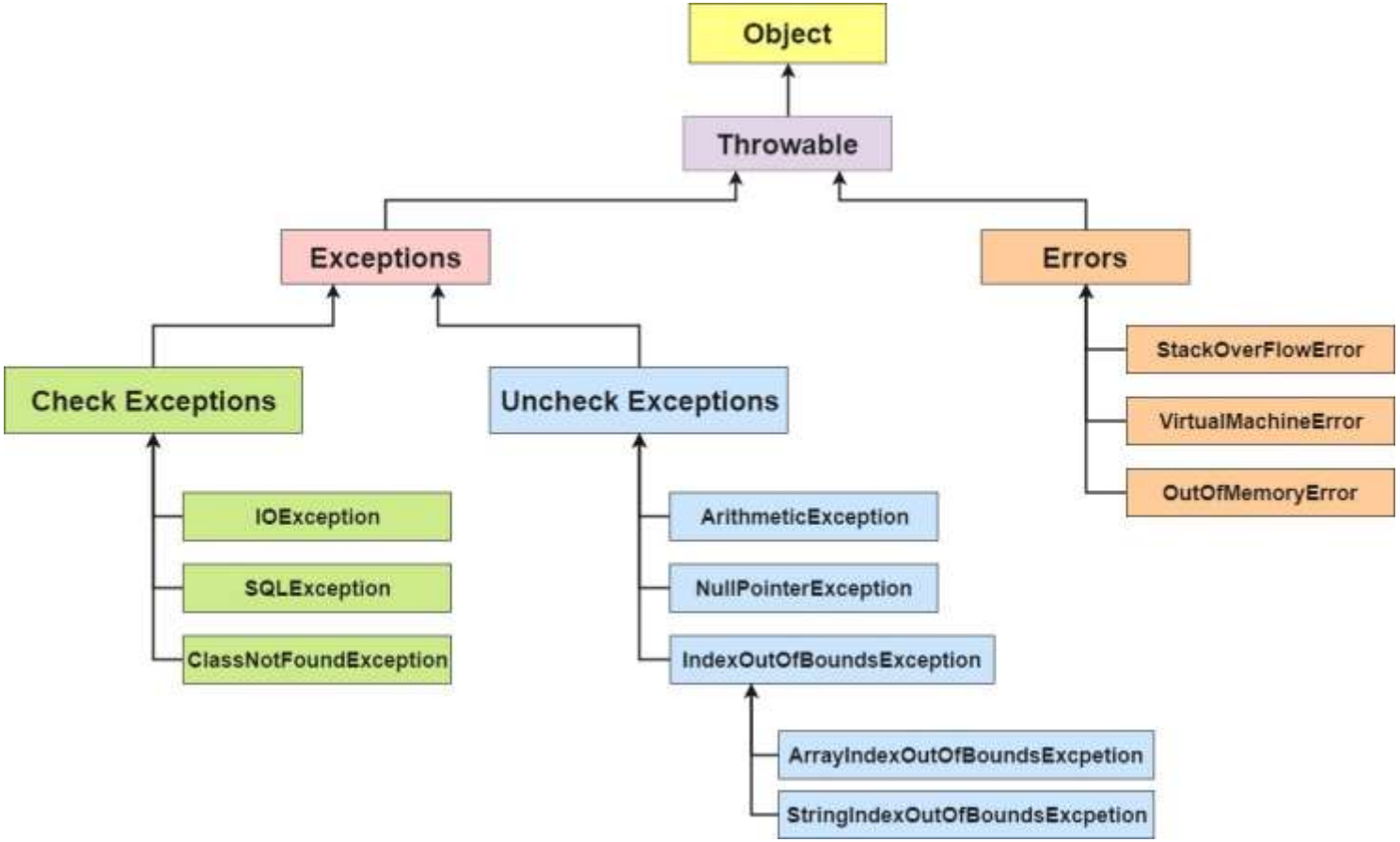


Image Source: https://simplesnippets.tech/exception-handling-in-java-part-1/

Complete List of Java Exceptions: https://programming.guide/java/list-of-java-exceptions.html

Uncaught Exceptions

```java
public class ExceptionUncaught {
    public static void main(String args[]) {
        int a = 10, b = 0;
        int c = a / b; // ArithmeticException: / by zero
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        String s = null;
        System.out.println(s.length()); // NullPointerException
    }
}
```

Caught Exceptions

```java
public class ExceptionCaught1 {
    public static void main(String args[]) {
        int a = 10, b = 0, c = 0;
        try {
            // try requires at least one catch or a finally clause
            c = a / b;
            System.out.println("This will never print");
        } catch (Exception e) { // ArithmeticException
            System.out.println("In Catch");
            System.out.println(e);
        } finally {
            // finally block is optional
            // finally block will always execute
            System.out.println("In Finally");
        }
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}
```

Caught Exceptions

```java
import java.util.Random;

public class ExceptionCaught3 {
    public static void main(String args[]) {
        int a = 10, b, c;
        Random r = new Random();
        for (int i = 1; i <= 32000; i++)
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345 / (b / c);
            } catch (ArithmeticException e) {
                System.out.println(e);
                a = 0;
            } finally {
                System.out.println(i + ": " + a);
            }
    }
}
```

try can be nested, please refer to **ExceptionTryNested.java**

Nesting Try Blocks

```
public class ExceptionTryNested {
    public static void main(String args[]) {
        int a = 10, b, c;
        Random r = new Random();
        try {
            for (int i = 1; i <= 32000; i++)
                try {
                    b = r.nextInt();
                    c = r.nextInt();
                    a = 12345 / (b / c);
                } catch (ArithmeticException e) {
                    System.out.println(e);
                    a = 0;
                } finally {
                    System.out.println(i + ": " + a);
                }
        } catch (ArithmeticException e) {
            System.out.println(e);
        }
    }
}
```

finally

```java
public class ExceptionCaught2 {
    public static void main(String args[]) {
        int a = 10, b = 0, c;
        try {
            c = a / b;
            System.out.println("This will never print");
        } catch (Exception e) { // ArithmeticException
            System.out.println("In Catch");
            System.out.println(e);
            return;
        } finally {
            // finally block will always execute
            System.out.println("In Finally");
        }
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}
```

Multiple catch clauses

```java
public class ExceptionMultipleCatch {
    public static void main(String args[]) {
        int a = 10, b = 0, c = 0;
        try {
            c = a / b;
        } catch (ArithmeticException e1) {
            System.out.println(e1);
        } catch (NullPointerException e2) {
            System.out.println(e2);
        } catch (Exception e) {
            System.out.println(e);
        } finally {
            System.out.println("In Finally");
        }
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}
```

catch(ArithmeticException | NullPointerException e)

catch(ArithmeticException | Exception e) - Error

throw

```java
public class ExceptionThrow {
    public static void f() {
        try {
            throw new NullPointerException("f");
        } catch(NullPointerException e) {
            System.out.println("Inside catch of f()");
            throw e; //rethrow the exception
        }
    }
    public static void main(String args[]) {
        try {
            f();
        } catch(NullPointerException e) {
            System.out.println("Inside catch of main()");
        }
    }
}
```

throws

```
1  public class ExceptionThrows {
2      public static void f() throws IllegalAccessException {
3          System.out.println("Inside f()");
4          throw new IllegalAccessException("f");
5      }
6
7      public static void main(String args[]) {
8          try {
9              f();
10         } catch (IllegalAccessException e) {
11             System.out.println("Inside catch of main()");
12             e.printStackTrace();
13         }
14     }
15 }
```

throws listing is not required for those of RuntimeException or any of their subclasses

Custom Exceptions

```
1   class MyException extends Exception {
2       private int detail;
3       MyException(int a) { detail = a; }
6       @Override
7       public String toString() { return "My Exception :  " + detail; }
10  }
11
12  public class ExceptionCustom {
13      static void compute(int a) throws MyException {
14          if (a > 10) {
15              throw new MyException(a);
16          }
17          System.out.println(a);
18      }
19      public static void main(String args[]) {
20          try {
21              compute( a: 10);
22              compute( a: 20);
23          } catch (MyException e) {
24              System.out.println(e);
25          }
26      }
27  }
```