**Programming in JAVA**

**What is a built-in function?**

Built-in functions refer to those pre-defined functions that come along with a programming language. They are intrinsic components of the language’s library, ready to be used without requiring any additional installation or importation. These functions have been designed to perform common tasks, thus facilitating efficient and swift coding.

**Purpose of built-in functions**

The main purpose of built-in functions is to enhance efficiency and productivity in programming. They render certain tasks easier and quicker to accomplish, thereby reducing development time. Moreover, they contribute to code readability and maintainability as they are standardized and well-documented.

**15 examples of built-in functions**

Let’s look into some specific examples of built-in functions across various programming languages.

1. **print()** – A commonly used built-in function in Python, print() outputs the specified message to the screen.
2. **len()** – This Python function returns the number of items in an object, like a list or a string.
3. **max()** – Another Python staple, max() extracts the maximum value from a list or array.
4. **abs()** – In Python, abs() provides the absolute value of a given number.
5. **type()** – This function in Python returns the data type of the object passed as an argument.
6. **parseInt()** – A JavaScript built-in function, parseInt() converts a string into an integer.
7. **isNaN()** – This JavaScript function checks if a value is NaN (Not a Number).
8. **encodeURI()** – In JavaScript, encodeURI() encodes a Uniform Resource Identifier (URI) by replacing each instance of certain characters with one, two, three, or four escape sequences representing the UTF-8 encoding of the character.
9. **alert()** – A simple JavaScript function, alert() creates a pop-up alert box with a specified message.
10. **charAt()** – This Java function returns the character at a specific index in a string.
11. **length()** – In Java, length() gives the length of a string.
12. **sqrt()** – A built-in function in C++, sqrt() calculates the square root of a number.
13. **pow()** – This C++ function computes the power of a number.
14. **exit()** – In C++, exit() terminates a program.
15. **tolower()** – This C++ function converts a character to lowercase.

**create your own exception class** (called a custom exception) when you want to handle application-specific errors in a more meaningful way than just using built-in exceptions like IOException or NullPointerException.

### Steps to Create Your Own Exception Class:

1. **Extend** either:
   * Exception → for **checked exceptions** (must be declared with throws and handled using try-catch).
   * RuntimeException → for **unchecked exceptions** (optional handling).
2. **Provide constructors**:
   * A no-argument constructor.
   * A constructor that accepts an error message.
   * Optionally, one that accepts both a message and a cause.

### Example: Custom Checked Exception

// Custom exception class

class InvalidAgeException extends Exception {

public InvalidAgeException(String message) {

super(message); // Call parent constructor

}

}

public class CustomExceptionExample {

// Method that throws the custom exception

static void validateAge(int age) throws InvalidAgeException {

if (age < 18) {

throw new InvalidAgeException("Age must be 18 or above to vote.");

} else {

System.out.println("Valid age. You can vote!");

}

}

public static void main(String[] args) {

try {

validateAge(15); // invalid age

} catch (InvalidAgeException e) {

System.out.println("Caught Exception: " + e.getMessage());

}

}

}

**Output:**

Caught Exception: Age must be 18 or above to vote.

### Example: Custom Unchecked Exception

// Custom runtime exception

class NegativeNumberException extends RuntimeException {

public NegativeNumberException(String message) {

super(message);

}

}

public class CustomRuntimeExample {

static int squareRoot(int num) {

if (num < 0) {

throw new NegativeNumberException("Number cannot be negative!");

}

return (int)Math.sqrt(num);

}

public static void main(String[] args) {

System.out.println(squareRoot(25));

System.out.println(squareRoot(-9)); // will throw exception

}

}

**Output:**

5

Exception in thread "main" NegativeNumberException: Number cannot be negative!

## 1. ****Garbage Collection in Java****

* **Definition:**  
  Garbage Collection (GC) in Java is the process of automatically reclaiming memory by destroying objects that are no longer reachable in the program.
* Unlike C/C++, where the programmer must explicitly free memory (free() or delete), in Java the **JVM handles memory cleanup**.
* The **Garbage Collector** keeps track of objects and removes those that no longer have any references.

👉 You can **suggest** garbage collection by calling:

System.gc();

but JVM may or may not run GC immediately.

## 2. ****finalize() Method****

* **Definition:**  
  The finalize() method is defined in the Object class.  
  It is called by the **Garbage Collector** before an object is destroyed, giving us a chance to release resources (like closing files, network sockets, etc.).

**Syntax:**

protected void finalize() throws Throwable {

// cleanup code

}

From Java 9 onward, finalize() is deprecated because it is **unpredictable** and not guaranteed to run. Instead, Java recommends using **try-with-resources** or AutoCloseable.

## 3. ****Example Program: Garbage Collection & finalize()****

class Demo {

int id;

Demo(int id) {

this.id = id;

System.out.println("Object " + id + " created");

}

// finalize() method

@Override

protected void finalize() throws Throwable {

System.out.println("Object " + id + " is garbage collected");

}

}

public class GarbageCollectionExample {

public static void main(String[] args) {

// Creating objects

Demo obj1 = new Demo(1);

Demo obj2 = new Demo(2);

// Making objects eligible for GC

obj1 = null;

obj2 = null;

// Request garbage collection

System.gc();

System.out.println("Main method ends");

}

}

### ****Possible Output (may vary because GC is not guaranteed immediately):****

Object 1 created

Object 2 created

Main method ends

Object 2 is garbage collected

Object 1 is garbage collected

# 1. ****Thread Class****

* **Definition:**  
  In Java, a thread can be created by **extending the Thread class** and overriding its run() method.  
  After creating a thread object, calling start() will internally call run().

### Example:

class MyThread extends Thread {

public void run() {

System.out.println("Thread is running using Thread class");

}

}

public class ThreadClassExample {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start(); // starts a new thread

}

}

**Output:**

Thread is running using Thread class

# 2. ****Runnable Interface****

* **Definition:**  
  Another way to create a thread is by **implementing the Runnable interface** and passing it to a Thread object.

### Example:

class MyRunnable implements Runnable {

public void run() {

System.out.println("Thread is running using Runnable interface");

}

}

public class RunnableExample {

public static void main(String[] args) {

MyRunnable r = new MyRunnable();

Thread t = new Thread(r);

t.start();

}

}

**Output:**

Thread is running using Runnable interface

# 3. ****Synchronization****

* **Definition:**  
  Synchronization in Java ensures that only **one thread at a time** can access a shared resource (to prevent data inconsistency).

# 4. ****Synchronized Methods****

* **Definition:**  
  A **synchronized method** ensures that only one thread can execute it at a time on the same object.

### Example:

class Table {

synchronized void printTable(int n) {

for (int i = 1; i <= 5; i++) {

System.out.println(n \* i);

try { Thread.sleep(500); } catch (Exception e) {}

}

}

}

public class SynchronizedMethodExample {

public static void main(String[] args) {

Table obj = new Table();

Thread t1 = new Thread(() -> obj.printTable(5));

Thread t2 = new Thread(() -> obj.printTable(10));

t1.start();

t2.start();

}

}

**Output (order may vary, but not mixed):**

5

10

15

20

25

10

20

30

40

50

# 5. ****Synchronized Block (Statement)****

* **Definition:**  
  Instead of synchronizing an entire method, you can use a **synchronized block** inside a method to lock only a specific part of code.

### Example:

class Printer {

void printMsg(String msg) {

synchronized (this) {

System.out.print("[" + msg);

try { Thread.sleep(500); } catch (Exception e) {}

System.out.println("]");

}

}

}

public class SynchronizedBlockExample {

public static void main(String[] args) {

Printer p = new Printer();

Thread t1 = new Thread(() -> p.printMsg("Hello"));

Thread t2 = new Thread(() -> p.printMsg("World"));

t1.start();

t2.start();

}

}

**Output (always proper brackets):**

[Hello]

[World]

# 6. ****Inter-thread Communication****

* **Definition:**  
  Threads can communicate using the methods wait(), notify(), and notifyAll().
  + wait() → releases lock and waits.
  + notify() → wakes up one waiting thread.
  + notifyAll() → wakes up all waiting threads.

### Example:

class Bank {

int balance = 10000;

synchronized void withdraw(int amount) {

System.out.println("Attempting to withdraw: " + amount);

if (balance < amount) {

System.out.println("Insufficient balance... waiting");

try { wait(); } catch (Exception e) {}

}

balance -= amount;

System.out.println("Withdrawal complete. Balance: " + balance);

}

synchronized void deposit(int amount) {

System.out.println("Depositing: " + amount);

balance += amount;

System.out.println("Deposit complete. Balance: " + balance);

notify();

}

}

public class InterThreadExample {

public static void main(String[] args) {

Bank bank = new Bank();

new Thread(() -> bank.withdraw(15000)).start();

new Thread(() -> bank.deposit(10000)).start();

}

}

**Output:**

Attempting to withdraw: 15000

Insufficient balance... waiting

Depositing: 10000

Deposit complete. Balance: 20000

Withdrawal complete. Balance: 5000

# 7. ****Deadlock****

* **Definition:**  
  Deadlock occurs when **two or more threads are waiting for each other’s lock** and none can proceed.

### Example:

class Resource {}

public class DeadlockExample {

public static void main(String[] args) {

final Resource r1 = new Resource();

final Resource r2 = new Resource();

Thread t1 = new Thread(() -> {

synchronized (r1) {

System.out.println("Thread 1: locked r1");

try { Thread.sleep(100); } catch (Exception e) {}

synchronized (r2) {

System.out.println("Thread 1: locked r2");

}

}

});

Thread t2 = new Thread(() -> {

synchronized (r2) {

System.out.println("Thread 2: locked r2");

try { Thread.sleep(100); } catch (Exception e) {}

synchronized (r1) {

System.out.println("Thread 2: locked r1");

}

}

});

t1.start();

t2.start();

}

}

**Possible Output (program may hang due to deadlock):**

Thread 1: locked r1

Thread 2: locked r2