__/__/____

OOP

- The principle of OOP are:

(a) Abstraction : Hiding complexity (object in java, car as an object not as collection of various parts.)

(b) Encapsulation : Binding code and data together and keep safe from outside interference (A class in java)

(c) Inheritance : Reuse of existing class/code (more features are added in subclass.)

(d) Polymorphism : Different behavior for different inputs (one interface shared by multiple methods)

Note: Comments : // (single line)
/* .....
*/ (multiple lines)

A simple Java program. (Example.java)

```
class Example {
    public static void main (String args[]) {    → main method
        System.out.println("Hello World");
    }
}
```

__/__/____

Note: public (for access all around the project)
void (returning nothing)

- Save the file as the class name itself.

- Java is simple, secure, portable, object-oriented, robust, multithreaded, architecture neutral, interpreted, high performance, distributed and dynamic.

- 1 byte = 8 bits

- The security and portability feature of JAVA comes from bytecodes. The JAVA compiler (javac) generates output for a virtual machine (JVM). The java interpreter executes the bytecodes and produces output.

- JVM differs platform to platform but all JVM understand the same bytecode.

Note: In structure/process oriented programming, we follow top-down approach. ~~whereas~~ Ex:- C
In object-oriented programming, we follow bottom-up approach. Ex:- JAVA.

- In the command prompt, compile the program with javac Example.java (which will create a class file for each class)

- To execute the program, java Example

- If multiple classes exist, java Example.classname.

☆ Datatypes

- Every variable has a type which is strictly defined.
- There are no automatical conversions of conflicting types like in C. It would result in error.
- Due to Java's portability requirement, all datatypes have strictly defined range. (size cannot vary)

— Integers

- All of these are signed. (both + and −)

byte	−	8	(−128 to 127)
short	−	16	(−32k to 32k)
int	−	32	(−2b to 2b)
long	−	64	

— Floating-Point

float	−	32	(single precision)
double	−	64	(double precision)

— Character

- Used to store characters (0 to 65k)
- In Java, char is 16-bit type.
- Instead of characters, ASCII values can also be given.

  88 → X

_/_/_

- They operate just like integers: (char ⊆ int)   ch1 = 'X'
  ch1++
  ch1 → 'Y'

– Boolean

- For logical values: true/false.

– Literals

- Used for creating constant values in Java.

- Integers → 1, 2, 42, 123_456_789 (, → _)
  Octal → 07, 05
  Hexadecimal → 0x (or) 0X
  Floating → 6.022E23 (6.022×10^{23})
  3.14f (or) 3.14F
  Boolean → true, false
  Character → 'X'

Note: \n → next line
\t → tab

☆ Operators

– Arithmetic

+, -, *, /, % (remainder), +=, -=, *=, /=, %=,

y = ++x → x = x+1, y = x (both ++)
y = x++ → y = x, x = x+1 (only x++)

- Bitwise

~ (NOT), | (OR), & (AND), ^ (XOR

- Java uses 2's complement to represent integers.

- Left shift (<<) and right shift (>>) are used to fill in the previous contents of the LSB (or) MSB (for signed numbers) to preserve sign of the value.

```
11111000  (-8)          0001000  (8)
  >>1                     >>1
1111100   (-4)          0000100  (4)
```

- Unsigned right shift (>>>) always shifts zeros into MSB.

- Relational

==, !=, >, <, >=, <= (result = boolean)

- Boolean Logic

&& (AND), || (OR), ! (NOT), ?: (ternary if-then-else)

```
if (a>b)
    max = a;
else
    max = b;
```

→ max = a>b ? a:b

__/__/__

Control Statements

- Condition is any expression that returns a boolean value. else clause is optional.

```
if (condition) statement1;
else statement 2;
```

-
```
switch (expression){
        case value1:
              break;      (used to terminate a statement sequence)
        case value 2:
              break;
              :
        default:
              .....
}
```

— LOOPS

→
```
while (condition){
        // body of loop
}
```
(Entry-controlled loop)

→
```
do {
        // body of loop
} while (condition);
```
(Exit-controlled loop)

→
```
for (initialization; condition; iteration){
        //body of loop
}
```

- return is used to exit program when ~~[illegible]~~ encountered.

___/___/___

✯ Math class

- Contains all floating-point functions used for geometry and trignometry.
- It defines two double constants : E (2.72)
  PI (3.14)
- static double sin(double arg) (returns sine of angle arg in radians)
- static double asin (double arg) (returns angle whose sine → arg)
- static double atan2 (double x, double y)
  (returns angle whose tangent is x/y)
- static double pow (double x, double y) (x^y)
- static double sqrt (double arg) ($\sqrt{arg}$)
- static int max (int x, int y) (returns greatest of x, y)
- static int min (int x, int y) (returns lowest of x, y)
- static double random()
- static double toRadians (double angle)
  static double toDegrees (double angle)
- static int round (float arg) (3.14 → 3)

__/__/__

✯ Arrays

- Group of like-typed variables referred by common name. They offer convenient means of grouping information.
- Declaration: type var-name[ ]; (or) type[ ] name;
- array-var = new type[size];

Ex:- month_days = new int [12]

- Assignment: array-var [index] = value
- To get the size of array, use array-name.length (length → instance variable)

✯ Classes & Objects

- Class is a template for an object
- Object is an instance of a class.
- Class defines a new data type which can be used to create objects of that type.

Ex:

```
class Box {
        double width;
        double height;
        double depth;
}
class BoxDemo {
        public static void main(String args[]){
                Box mybox = new Box();   (object)
```

```
        double vol;
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;
        vol = mybox.width * mybox.height * mybox.depth;
        System.out.println("Volume is " + vol);
    }
}
```

☆ **Constructors**

- A constructor initializes an object immediately upon creation. It has same name as class. Once defined, constructor is automatically called after object created.

```
class Box {
    double width;
    double height;
    double depth;
Box () {
    System.out.println("Constructs Box");
    width = 10;
    height = 20;
    depth = 30;
}
    :
Box mybox1 = new Box()   (call constructor)
```

– "**This**" **Keyword**

- It can be used inside any method to refer to the

current object.

```
Box (double w, double h, double d){
    this.width = w;
    this.height = h;
    this.depth = d;
}
```

- Garbage Collection

• "new" operator is used to dynamically allocate object in the memory.

• When no references to an object exist, that object is said to be no longer needed and needs to be deallocated

```
Runtime r = Runtime.getRuntime();
r.gc();
```

- Finalize() Method

• Java runtime calls finalize() method whenever it is about to recycle an object of that class.

• Inside this method, we will specify those actions to be performed before an object is destroyed.

```
protected void finalize()
{
    // finalization code
}
```

- Stack

```
class Stack {
    int stck[] = new int[10];
    int tos;
    Stack() {tos = -1;}
    void push (int item) {
        if (tos == 9)
            System.out.println("Stack is full");
        else
            stck[++tos] = item
    }
    int pop () {
        if (tos < 0) {
            System.out.println("Stack underflow");
            return 0;
        }
        else
            return stck[tos--];
    }
}
class TestStack {
    public static void main (String args[]) {
        Stack mystack1 = new Stack();
        for (int i=0; i<10, i++)
            mystack1.push(i);
        System.out.println("Stack in mystack :");
        for (int i=0, i<10, i++)
            System.out.println(mystack1.pop());
    }
}
```

__/__/__

☆ Overloading (Polymorphism)

- In Java, it is possible to define two or more methods of the same name within the same class, as long as they differ by types, parameters or return types. (Method overloading.)

Ex:

```
class OverloadDemo{
    void test(){
        System.out.println("No parameters");
    }
    void test (int a, int b){
        System.out.println("a,b = "+a+b);
    }
    double test (double a){
        System.out.println("double a : "+a);
        return a*a ;
    }
}
class Overload {
    public static void main (String args[]){
        OverloadDemo ob = new OverloadDemo();
        double result ;
        ob. test()                          // No parameters
        ob. test (10, 20)                   // a,b : 10 20
        result = ob. test (123.45);         // double a : 123.45
        System.out.println("Result : "+ result);
                                            // result : 15239.9025
    }
}
```

Note: In some cases, when overload method is called and the given parameters do not match the type mentioned, automatic type conversion takes place to convert the parameters to desired type.

- Overloading can also be done for constructors.

☆ Objects as Parameters

```
class Test {
    int a, b;
    Test (int i, int j) {
        a = i;
        b = j;
    }
    boolean equals (Test o) {
        if (o.a == a && o.b == b)
            return true;        -> parameter object
        else
            return false;
    }
}

class PassOb {
    public static void main (String args[]) {
        Test ob1 = new Test (100, 22);
        Test ob2 = new Test (100, 22);
        Test ob3 = new Test (-1, 1);
        System.out.println ("ob1.equals (ob2)); (true)
        System.out.println (ob1.equals (ob3)); (false)
    }
}
```

__/__/____

Note: Objects can also be returned by a method.

☆ Recursion

- Any method that calls itself is said to be recursive.

```
class Factorial {
    int fact (int n) {
        int result;
        if (n==1)
            return 1;
        result = fact(n-1) * n;
        return result;
    }
}
```

☆ Access Control

- Through encapsulation, you can control what parts of a program can access members of a class, to prevent misuse.

- public (can be accessed throughout program)
  private (restricted to a certain method / class),
  protected (accessible within same class ~~in same~~ and other classes in same package)

☆ static Keyword

- When a member is declared static it can be accessed before any objects of it's class are created without reference to any object. Ex:- main();

☆ final Keyword

- A variable declared as ~~final~~ final prevents any further modification of it's contents. (constant)

☆ Nested Classes

- Inner (nested) class can access all members of outer class while outer (enclosing) class cannot access members of inner class.

- Inner class is known to outer class but not outside of outer class.

```
class Outer {
    int outer_x = 100;
    void test () {
        Inner inner = new Inner();
        inner.display();
    }
    class Inner {
        int y = 10;
        void display (){
            System.out.println(outer_x);    (100)
        }
    }
    void show_y(){
        System.out.println(y);    (Error)
    }
}
```

```
class Demo {
    public static void main(String args[]){
        Outer outer = new Outer();
        outer.test();
    }
}
```

☆ String Class

- Every string created in Java is an object of type String.
- Objects of type String are immutable unless a peer class called StringBuffer is used.
- s1.length()
  s1.equals(s2)
  s1.indexOf('b')
  s2 = s1.replace('x','y')
  s2 = s1.toLowerCase()
  s2 = s1.toUpperCase()

☆ Variable No. of Arguments

```
class A {
    static void vtest(int... v){
        System.out.println("Number of args: "+v.length);
        for(int x : v)          -> for each
            System.out.println(x + " ");
        System.out.println();
    }
    public static void main(String args[]){
        vtest(1,2,3,4,5);
    }
}
```

__/__/__

Note: for (int i : array) ⇌ for (int i=0; i<array[i]; i++)
//print i //print

☆ Inheritance

- You can create a general class that defines traits common to a set of related items, which can be inherited by other, more specific classes, each having things unique for it.

- In Java, inherited class → superclass
  inheriting class → subclass.

- Subclass can inherit all methods and variables of superclass using extends keyword, but not vice-versa

  class subclass extends superclass {...}

– Properties

- Subclass cannot access private members of superclass.

- Subclass constructor can ~~be~~ call superclass constructor using super(), else, no-argument constructor called.

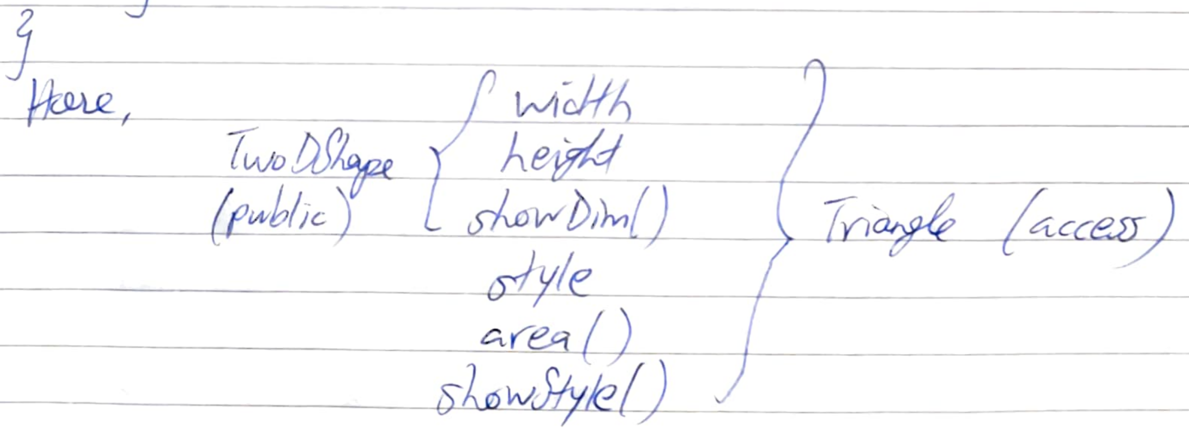
- Each ~~subclass~~ class can have MAX one superclass; Each superclass can have MULTIPLE subclasses.

- A
  B extends A
  C extends B is also possible. (hierarchy)

_/_/_

Ex:

```
class TwoDShape {
    double width;
    double height;
    void showDim(){
        (Print width & height)
    }
}
class Triangle extends TwoDShape {
    String style;
    double area(){
    double return width * height/2;
    }
    void showStyle(){
        (Print style)
    }
}
```



- Super keyword

- It is a reference variable to parent class objects.
- It can be used to invoke parent class variables, methods and constructors.

__/__/____

- Variables : super.variableName (parent and child class must have same fields, names)
- Methods : super.methodName();
- Constructors : super(); (with parameters if necessary)
- Super() must always be the first ~~element~~ statement inside the subclass constructors.
- In a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.
- A superclass reference can refer to a subclass object (copy of superclass)

— Method Overriding (polymorphism)

- It is a technique in which a method in parent class is redefined or overridden in child class.
- When overridden method is called in subclass, it will only refer to subclass version of that method, and version defined by superclass will be hidden.
- Overridden methods have same nature of return-type ~~and~~ parameters
- To access superclass version, ~~used~~ use super.methodName();

__/__/____

- Dynamic method dispatch is a ~~meha~~ mechanism in which a call to an overridden method is resolved at run time rather than compile time.

- When an overridden method is called through superclass reference, Java determines which version of that method to ~~refer to~~ execute based upon type of object.

Exa

```
class Sup {
    void who(){
        System.out.println("who() in Sup");
    }
}
class Sub1 extends Sup {
    void who(){
        System.out.println("who() in Sub1");
    }
}
class Sub2 extends Sup {
    void who(){
        System.out.println("who() in Sub2");
    }
}
class DynDispDemo {
    public static void main (String args[]){
        Sup superOb = new Sup();
        Sub1 subOb1 = new Sub1();
        Sub2 subOb2 = new Sub2();
        Sup supRef    //reference
        supRef = superOb;
        supRef.who();
```

```
        supRef = subOb1;
        supRef.who();
        supRef = subOb2;
        supRef.who();
    }
}
```

→ Output:
```
who() in Sup
who() in Sub1
who() in Sub2
```

- Superclasses and subclasses form a hierarchy that moves from lesser to greater specialization.
- Superclass provides all methods that subclass can use directly, while also defining methods on it's own.

— Abstract Classes

- It is the process of hiding implementation details and showing only functionality to user. (partial implementation)
- In case of abstract method, you must leave off body of method, and placed under abstract class.
- In case of abstract class, it is only partially implemented and left to subclasses for further implementation.

Ex:
```
abstract class Shape {
        abstract void draw();
}
class Rectangle extends Shape {
```

```
    void draw() { (Print "Drawing rectangle") }
}
class testAbstraction1 {
    public static void main(String args[]) {
        Shape s = new Rectangle()  // abstract reference
                                      variable.
        s.draw();
    }
}
```

Final Keyword

- If you do not want class to be subclassed, use final keyword. It can also be used to avoid overriding methods and make values of variables constant.

Object Class

- It is a parent class of all classes in Java by default (topmost class) present in java.lang package.

- A variable of type Object can refer to an object of any other class, including an array.

Object clone() → creates a copy
boolean equals(Object obj) → test whether two objects are equal
void finalize() → called before recycling object

☆ Interface

- It is an abstract way of defining what a class must do, but not how to do it.

- They are syntactically similar to classes, but with public methods w/o body and public final static variables. (constants)

- Using interfaces, Java allows to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Imp · Once an interface has been defined, one or more classes can implement that interface.

- If a class implements more than one interface, the interfaces are seperated with a comma.

Ex:
```
interface Series {
        int getNext();
        void reset();
        void setStart(int x);
}
class ByTwos implements Series {
    int start;
    int val;
    ByTwos(){
        start = 0;
        val = 0;
    }
    public void setStart(int x){
        start = x;
```

__/__/____

```
        val = x;
    }
    public int getNext(){
        val += 2;
        return val;
    }
    public void reset(){
        val = start;
    }
}
public class ByTwosSeries{
    public static void main(String[] args){
        ByTwos ob = new ByTwos();
        System.out.println("Series starting at: "+ob.val);
        for(int i=0; i<5; i++)
            System.out.println("Next value: "+ob.getNext());
        ob.setStart(100)
        System.out.println("\n Starting at: "+ob.val);
        for(int i=0; i<5; i++)
            System.out.println("Next value: "+ob.getNext());
    }
}
```

→ <u>Output</u>

Series Starting at 0	Starting at : 100
Next Value : 2	Next Value : 102
Next Value : 4	104
6	106
8	108
10	110

_ / _ / _

- It is possible to refer to any instance of any class that implements the declared interface by using reference variables

Ex:-
```
Series iRef;
  for (int i=0; i<5; i++){
      iRef = ob2;       <- byTwos
      (Print iRef.getNext());
      iRef = ob3;       <- byThrees
      (Print iRef.getNext());
  }
```

Ex: (To create a Dynamic Stack)
```
interface IntStack {
      void push (int item);
      int pop ();
      boolean isEmpty ();
      boolean isFull();
}
class DynStack implements IntStack {
     public void push (int item){
         if (isFull()){
             int temp[] = new int[stck.length *2];
             for (int i=0; i<stck.length; i++)
                 temp[i] = stck[i];
             stck = temp;
             stck[++tos] = item;
         }
         else
             stck[++tos] = item;
     }
```
(Rest all methods are same as in normal stack)

__/__/___

```
class Test2 {
    public static void main(String args[]){
        DynStack myStack1 = new DynStack(5);
        for (int i=0; i<12; i++)
            myStack1.push(i);
        System.out.println("Stack in myStack1: ");
        for (int i=0; i<12; i++)
            System.out.println("myStack1.pop());
    }
}
```

- Interfaces can inherit another using the <u>extends</u> keyword.
- All variables and methods are public if interface itself is declared ~~public~~ public.
- An interface can be declared a member of another interface of a class, called nested interface, which can be declared public, private or protected.

<u>Classes</u>	<u>Interface</u>
• It can be instantiated	• Cannot be instantiated
• Does not support multiple inheritance	• Does support multiple inheritance
• Can be inherited from another class using <u>extends</u>	• Can be ~~in~~ inherited by class using <u>implements</u> ~~and~~ by an interface using <u>extends</u>

• Variables can be static, final or neither.	• All variables are static and final.
• Variables and methods can be declared using any access specifiers.	• All variables and methods are declared as public.

—x—

☆ Packages

- They act as containers for classes, providing a way of organizing your code and of controlling access to the code.
- Classes in packages can be easily reused.
- Two different packages can contain classes with same name.
- Packages participate in Java's access control mechanism by providing a way to 'hide' classes from other programs or packages.
- In file system, each package is stored in it's own directory that has same name as package.
- For nested packages : package pkg1.pkg2
- To compile, javac pkg1/pkg2/MyClass.java
- To run, java pkg1.pkg2.MyClass

Package & Member Access

	Private	Default	Protected	Public
Visible within same class	✓	✓	✓	✓
Visible within same class package by subclass	X	✓	✓	✓
Visible within same package by non-subclass	X	✓	✓	✓
Visible within diff packages by subclass	X	X	✓	✓
Visible within diff packages by non-subclass	X	X	X	✓

Importing

- import statement brings certain classes or entire package into visibility.
- pkg1.pkg2.MyClass v = new pkg1.pkg2.MyClass();
- import pkg.classname
- java.lang contains a lot of general-purpose classes (imported automatically)
  java.io contains input/output classes.
  java.net contains classes supporting networking.
  java.util contains utility classes.
- import static lets you import static members of class. (Math.sqrt() ⟶ sqrt())

- After saving package file, to compile it:
  javac simple.java
  javac -d . Simple.java
  destination ↲ ↳ same directory

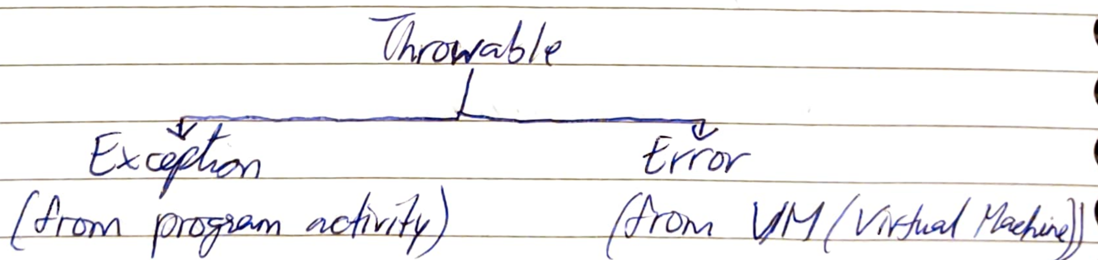
- To run, java mypack.Simple

Exceptions

- It is an abnormal (error) condition that arises in a code sequence at runtime, which causes system to crash and affect other programs running in parallel.

- When an exception condition arises, an object representing the exception is created and thrown in the method that caused the error.

- All exception classes are derived from Throwable class.



- In Java, exception handling is managed via try, catch, throw, throws and finally.

– try & catch

try → monitors for errors in code
catch → handles exceptions

__/__/____

- If the program does not catch the exception, then it will be caught by JVM itself, terminating the execution and displaying stack trace and error message.

```
class ExcDemo {
    public static void main(String args[]){
        int numerator = {4, 8, 16, 32, 64, 128, 256, 512};
        int denominator = {2, 0, 4, 4, 0, 8};
        for (int i=0; i<numerator.length; i++){
            try {
                System.out.println(numerator[i]+ "/" +
                            denominator[i] + "is" +
                            numerator[i]/denominator[i]);
            }
            catch (ArithmeticException exc){   ← object
                System.out.println("Can't divide by zero!");
            }
            catch (ArrayIndexOutOfBoundsException exc) {
                System.out.println("No matching elements found");
            }
        }
    }
}
```

→
```
4/2 is 2
Can't divide by zero
16/4 is 4
32/4 is 8
Can't divide by zero
128/8 is 16
No matching element found
No matching element found.
```

- When the exception is handled by the inner try block in nested try blocks, the program will continue to execute. However if the outer try block catches the exception, the program will terminate.

— Throwing an Exception

- It is possible to manually throw an exception by:

```
throw new ArithmeticException();
```

- An exception caught by one catch ~~exception~~ statement can be rethrown so that it can be ~~catc~~ caught by the outer/next catch statement. It cannot be recaught by the same catch statement.

- printStackTrace() is a method defined by Throwable that can display the standard error message and the record of the method calls that led to exception.

- toString() is a method called when exception is used as an argument to println, to retrieve standard error msg.

```
System.out.println(exc);   ← toString()
exc.printStackTrace();     ← detailed message
```

— finally block

- Used to terminate current block of code/methods after try/catch block. (exit block)

__/__/__

Built-In Exceptions

- Unchecked exceptions: Happen at runtime when executable program starts running. Need not be included in throws list.

- Checked exceptions: Happen at compile time when source code is transformed into executable code. Must be included in throws list.

- ArithmeticException (arithmetic error such as integer/0)
~~ArrayOutOfB~~
ArrayIndexOutOfBounds (index of array out-of-bounds limits)
ArrayStoreException (assignment of incompatible type of array element)
IndexOutOfBoundsException (some type of index out-of-bounds)
NegativeArraySizeException (array created with (-) size)
NullPointerException (invalid use of null reference)
NumberFormatException (invalid conversion of string → number)
StringIndexOutOfBoundsException (attempt to index out of bounds of string size)
TypeNotPresentException (type not found)
ClassNotFoundException (class not found)
IllegalAccessException (access to class is denied)

Creating Exception Subclasses (Custom)

```
class NonIntResultException extends Exception {
    int n;
    int d;
```

```
NonIntResultException (int i, int j) {
        n = i;
        d = j;
    }
    public String toString() {
        return "Result of " + n + " / " + d + " is not integer.";
    }
}
class CustomExceptDemo {
    public static void main (String args[]) {
        int numerator[] = {4, 8, 15, 32, 64, 127, 256, 512};
        int denominator[] = {2, 0, 4, 4, 0, 8};
        for (int i=0; i<numerator.length; i++) {
            try {
                if ((numerator[i]%2) != 0)
                    throw new NonIntResultException (numerator[i],
                                                     denominator[i]);
                System.out.println (numerator[i] + " / " + denominator[i]
                                    + " is " + numerator[i]/denominator[i]);
            }
            catch (ArrayIndexOutOfBoundsException exc) {
                System.out.println (" No matching element found ");
            }
            catch (NonIntResultException exc) {
                System.out.println (exc);
            }
        }
    }
}
```

→ 4/2 is 2
Can't divide by zero
Result of 15/4 is non-integer
32/4 is 8
Can't divide by zero
Result of 127/8 is non-integer
No matching element found
No matching element found.

__/__/__

Generics

- Generics creates classes, methods and interfaces in which the type of data on which it operates is specified as a parameter.

- It adds the type safety feature to make typecasting automatic and implicit, so that it can convert Object to actual type of data being operated on.

- Major advantage is it provides high level of reusability

→ W/O Generics

```
class KVPair {
    Object key, value;
    Object getKey(){ return key;}
    Object getValue() { return value;}
    void setKey(Object ob) { key = ob;}
    void setValue(Object ob) { ~~key~~ value = ob;}
}
public class TestGenerics {
    public static void main(String[] args){
        KVPair pair = new KVPair();
        pair.setValue("Address");
        String v = (String) pair.getValue();
        System.out.println("Value: "+v);
    }
}
```

→ <u>With Generics</u>

```
class KVPair <T> {          ← type parameters
    T key, value;
    T getKey() { return key; }
    T getValue() { return value; }
    void setKey(T ob) { key = ob; }
    void setValue(T ob) { value = ob; }
}
public class TestGenerics {
    public static void main(String[] args) {
        KVPair pair<String> pair = new KVPair();
        KVPair<String> pair = new KVPair<>();
        pair.setValue("Address");
        String v = pair.getValue();   // no typecast
        System.out.println("Value: " + v);
    }
}
```

- Here T is a placeholder for an actual type, which is provided when a KVPair object is created.

— <u>Methods in Object class</u>

- Object clone() → creates a copy of this object.
- boolean equals(Object obj) → test equality.
- Class< > getClass() → return class of object
- String toString() → return string describing object.

Wrapper Class

- It is a class whose object wraps/contains primitive data types
- The wrapper class in Java provides the mechanism to convert primitive ⇄ object

Primitive	Wrapper
boolean	Boolean
char	Character
int	Integer

- Generic types differ based on their type arguments.
- Generic methods take a type parameter cited by actual type and returns some value after performing a task.
- Generic classes are implemented just like normal class with the difference being it contains a type parameter section. There can be multiple type of parameters seperated by comma <T, P>.

Generic Stack Class

```
class StackFullException extends RuntimeException {
    private static final long serialVersionUID = 1L;
    public StackFullException (String message) {
        super (message);
    }
}
```

__/__/____

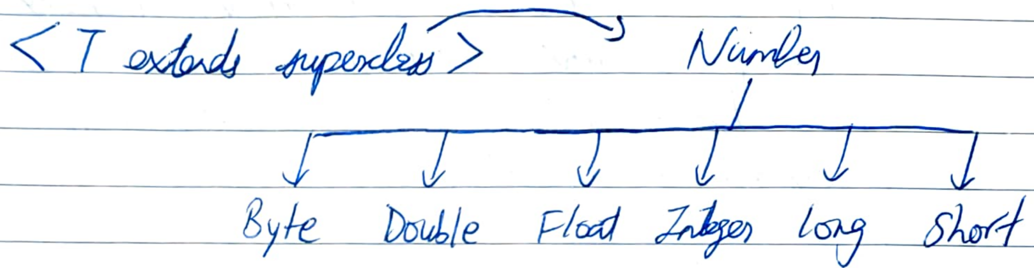
```java
class StackEmptyException extends RuntimeException {
    private static final long serialVersionUID = 1L;
    public StackEmptyException (String message) {
        super (message);
    }
}
public class generics3 {
    public static void main (String[] args) {
        Integer[] arr = new Integer[5];
        Stack <Integer> stack = new Stack<>(5, arr);
        try {
            stack.push (10);
            stack.push (20);
            stack.push (30);
            stack.push (40);
            stack.push (50);
            stack.push (60);        // Overflow
        }
        catch (StackFullException e){
            System.out.println("e.getMessage());
        }
        try {
            while (! stack.isEmpty()) {
                System.out.println ("Popped : " + stack.pop());
            stack.pop ();           // Underflow
        } catch (StackEmptyException e) {
            System.out.println("e.getMessage());
        }
    }
}
```

```
class Stack <T> {
    private int size;
    public T[] stackAr;
    private int top;
    public Stack (int size, T[] arr) {
        this.size = size;
        stackAr = arr;              //Init
        top = -1;
    }
    public void push (T value) {
        if (isFull ()) {
            throw new StackFullException ("Cannot push"
                          + value + "Stack is full");
        }
        stackAr[++top] = value;
    }
    public T pop() {
        if (isEmpty ()) {
            throw new StackEmptyException("Stack is empty");
        }
        return stackAr[top--];
    }
    int returnSize() {
        return size;
    }
    public boolean isEmpty () {
        return (top == -1);
    }
    public boolean isFull () {
        return (top == size - 1);
    }
}
```

__/__/__

Bounded Types

- To limit the type parameter, Java provides bounded types. Here, we can create an upper bound that declares the superclass from which all type arguments can be derived.



- For class X<T, V extends T>,
  ~~X<Integer,~~
  X<Number, Integer> x = new X<Number, Integer>(10.4, 12);
  is valid.

- Wildcard arguments are unknown type arguments that can hold any type of objects.

Ex: Stack<?>

→
```
boolean absEqual(NumericFns<?> ob) {
    if (Math.abs(num.doubleValue()) ==
          Math.abs(ob.num.doubleValue()))
        return true;
    return false;
}
```

_/_/_

Generic Methods

→ public <T> void methodName (T parameter) { }

```
public class GenericMethodTest {
    public static <E> void printArray(E[] inputArray) {
        for (E element : inputArray)
            System.out.print(" " + element);
        System.out.println();
    }
    public static void main (String[] args) {
        Integer[] intArray = {1, 2, 3, 4, 5};
        Character[] charArray = {'H', 'E', 'L', 'L', 'O'};

        printArray (intArray);
        printArray (charArray);
    }
}
```

Generic Constructors

- Constructors can be generic, even though it's class is not.

```
<T extends Number> Summation (T arg) {
    sum = 0;
    for (int i=0; i<= arg.intValue(); i++)
        sum += i;
}
```

Inheritance & Interfaces

```
class sub<T, V> extends sup<T> { }
```

- Even, class C<T> implements I<T> { }

Restrictions

- Static variables & methods cannot use type parameters declared by their generic class. (static T x;)

- You cannot create an instance of a type parameter.

```
T x;
x = new T();   ✗
```

- You cannot create an array whose element type is the type parameter nor an type-specific generic reference.

```
class C<T>{
    T[] x;
    x = new T[10];   ✗
    C<String>[] data = new C<String>[10] //specific   ✗
    C<?>[] data = new C<?>[10];   ✓
```

- You cannot create generic exception classes (cannot extend Throwable)

__/__/____

Multithreading

- A multithreaded program contains two or more parts that can run concurrently.
- Here, each part of the program is called a thread, each of which define a seperate path of execution.
- → Process-based multitasking allows your computer to run two or more programs concurrently. Here, a program is the smallest unit of code that can be dispatched by the scheduler.
- Thread-based multitasking allows your computer to perform two or more tasks simultaneously. Here, a thread is the smallest unit of dispatchable code.
- Processes are heavyweight tasks that require their own address space. Inter-process communication is expensive and limited. So is the context switching
- Threads, however, are lightweight, share the same address space, inter-thread communication is inexpensive
- Multithreading enables us to write very efficient programs making maximum use of CPU, minimizing idle time.
- → In a singled-threaded environment, when a thread blocks (or suspends execution) while waiting for some resource, the entire program stops running

__/__/____

- On the other hand, in a multi-threaded environment, only the single thread that is blocked pause, the other threads continue to run.

- A thread can be running, suspended, resumed, blocked while waiting for a resource and terminated.

- In Java, you can create a thread by either:
  (a) extends Thread
  (b) implements Runnable

- getName() → returns name of thread.
  getPriority() → returns priority of thread.
  isAlive() → checks if thread is still running.
  join() → waits for thread to terminate.
  run() → entry point for thread
  sleep() → temporary suspension (for certain milliseconds)
  start() → calls run method.

→
```
class C extends Thread {
    public void run() {
        for (int k = 1; k <= 5; k++) {
            System.out.println("In From Thread C: k = " + k);
        }
    }
}

class Test {
    public static void main (String[] args) {
        C thread3 = new C ();
        thread3.start();
```

_/_/_

- While implementing Runnable, in the main function:

```
C obj_1 = new C();
Thread thread1 = new Thread(obj_1);
thread1.start();
```

- Thread constructors:

```
Thread(Runnable threadOb, String name);
```

- isAlive() is a boolean method to check if thread is still running.
- join() waits until thread on which it is called terminates.

Thread Priorities

- Used by thread scheduler to decide when each thread should be allowed to run, get more CPU time
- To set a threads Priority, use setPriority() method
- A high priority thread can prempt a low-priority one.

MIN_PRIORITY → 1
NORM_PRIORITY → 5
MAX_PRIORITY → 10

PTO →

__/__/__

✱ Synchronization

- When multiple threads need access to a shared resource, they need some way to ensure that the resource will be used by one thread at a time.

- Monitor is an object that is used as a lock. Only one thread can own a monitor at a time. When a thread enters a monitor, the other threads are suspended until it exits the monitor.

- ```
  synchronized (object) {
                              ← Synchronized block
  }
  synchronized static methodName() {   ← Synch. method
  ```

✱ Interthread Communication

- wait() → tells calling thread to give up monitor and go to sleep until other thread notifies
  notify() → wakes up first thread that called wait()
  notifyAll() → wakes up all threads that called wait(). Highest priority thread will run first.

→
```
class multithreading { {
    public static void main(String[] args) throws
                                   InterruptedException {
        calc Calculator = new calc();
        Calculator.start();
```

PTO→
```

```
        synchronized (Calculator) {
            System.out.println("Calling wait method");
            Calculator.wait();
            System.out.println("Got notification");
        }
        System.out.println("Got notification");
      }
      System.out.println("Total: "+ Calculator.total);
    }
}

class calc extends Thread {
    int total = 0;
    public void run() {
        synchronized (this) {
            System.out.println("Starting calculation");
            for (int i=0; i<=1000; i++) {
                total += i;
            }
            System.out.println("Giving notification call");
            notify();
        }
    }
}
```

OUTPUT

```
Calling wait method
Starting calculation
Giving notification call
Got notification
Total: 500500
```

__/__/____

Deadlock

- It occurs when two or more threads are unable to proceed because each is waiting for the other to release a lock, causing circular dependency.

- For example, Thread 1 first acquires lock of Object X
  then tries to acquire lock of Object Y
  Thread 2 first acquires lock of Object Y
  then tries to acquire lock of Object X
  Thread 1 cannot proceed and waits for Thread 2
  to release Object Y
  Thread 2 cannot proceed and waits for Thread 1
  to release Object X.

Other Thread Controls

suspend() → pauses the thread execution until resumed
resume() → resumes execution of suspended thread
stop() → stops the thread, cannot be resumed.

Conclusion

- Key to utilizing multithreading is to think concurrently rather than serially.

- More threads → More CPU time → degrades performance of program.