Unit - 4

Packages

Exception Handling

String Handling

Packages

- A package as the name suggests is a pack (group) of classes, interfaces and other packages. In java we use packages to organize our classes and interfaces. We have two **types of packages in Java**: built-in packages and the packages we can create (also known as user defined packages).
  - java.util.Scanner
    - Here: **java** is a top level package
      util is a sub package
      and **Scanner** is a class which is present in the sub package **util**.
- Classes defined within a package must be accessed through their package name.

Advantages of using a package in Java

- **Reusability**
  - Using packages, you can create such things in form of classes inside a package and whenever you need to perform that same task, just import that package and use the class.
- **Better Organization**
  - Large java projects often have several hundreds of classes, it is always required to group the similar types of classes in a meaningful package name so that you can organize your project better.
- **Name Conflicts**
  - We can define two classes with the same name in different packages. This avoids name collision.

Steps in creating a package

- Create a package(folder) named **p1**.
- Create a file named **MyCalculator.java** inside **p1** package.
- Write the code in **MyCalculator.java** file.
- Create a file named Create a file named **TestPackage.java** outside **p1** package.

Packages and Member Access

- The visibility of an element is determined by its access specification – private, public, protected or default – and the package in which it resides.
  - Thus, the visibility of an element is determined by its visibility within a class and its visibility within a package.

Packages and Member Access

	private	default	protected	public
Visible within same class	Yes	Yes	Yes	Yes
Visible within same package by subclass	No	Yes	Yes	Yes
Visible within same package by non-subclass	No	Yes	Yes	Yes
Visible within different package by subclass	No	No	Yes	Yes
Visible within different package by non-subclass	No	No	No	Yes

private – within the package

```
package packaccess;
class PrivateClass{
     private int x= 10;
     private void show(){
          System.out.println("Private class");
  }
}
public class PackAccess {
   public static void main(String[] args) {
       PrivateClass obj=new PrivateClass();
       System.out.println(obj.x); //Compile Time Error
       obj.show(); //Compile Time Error
   }
}
```

private – outside package by subclass

```
package packaccess;
public class PackAccess {
     private int x= 10;
     private void show(){
          System.out.println("Private class");
  }
}
```

```
import packaccess.PackAccess;
class InheritPack extends PackAccess {
   void display() {
      System.out.println(x);
      show();
   }
}
```

private – outside the package

```
package packaccess;
public class PackAccess {
     private int x= 10;
     private void show(){
          System.out.println("Private class");
  }
}
```

```
package packaccdemo;
public class PackAccDemo {
   public static void main(String[] args) {
       packaccess.PackAccess ob1 = new packaccess.PackAccess();
       System.out.println(ob1.x);
       ob1.show();
   }
}
```

default – within the package by subclass

```
package packaccess;
public class PackAccess {
     int x= 10;
     void show(){
          System.out.println("Private class");
  }
}
```

```
class Test extends PackAccess {
   void display() {
       PackAccess obj = new PackAccess();
       System.out.println(obj.x); //Compile Time Error
       obj.show(); //Compile Time Error
   }
}
```

default – within the package non-subclass

```
package packaccess;
class PrivateClass{
     int x= 10;
     void show(){
          System.out.println("Private class");
  }
}
public class PackAccess {
   public static void main(String[] args) {
       PrivateClass obj=new PrivateClass();
       System.out.println(obj.x); //Compile Time Error
       obj.show(); //Compile Time Error
   }
}
```

default – outside the package

```
package packaccess;
public class PackAccess {
    int x= 10;
    void show(){
        System.out.println("Private class");
  }
}
```

```
package packaccess;
public class PackAccess {
    int x= 10;
    void show(){
        System.out.println("Private class");
  }
}
```

default – outside package by subclass

```
package packaccess;
public class PackAccess {
     int x= 10;
     void show(){
          System.out.println("Private class");
  }
}
```

```
package packaccdemo;
import packaccess.PackAccess;
class InheritPack extends PackAccess {
   void display() {
      System.out.println(x);
      show();
   }
}
```

protected – within class, subclass in same package, non-subclass in same package

```
package packaccess;
class Testing {
    protected int y= 100;
    protected void show(){
        System.out.println("Good Day!!");
  }
}
public class PackAccess {
    protected int x= 10;
    protected void show(){
        System.out.println("Hello World!");
    }
  public static void main(String[] args) {
      Test obj=new Test();
      System.out.println(obj.x);
      obj.show();
      PackAccess ob1 = new PackAccess();
      System.out.println("Within class: " + ob1.x);
      ob1.show();
      Testing ob2 = new Testing();
      System.out.println("Same package, non subclass: " + ob2.y);
      ob2.show();
   }
}
```

```
class Test extends PackAccess {
   void display() {
      PackAccess obj = new PackAccess();
      System.out.println(obj.x);
      obj.show();
   }
}
```

protected – different package subclass, different package non-subclass

```
package packaccdemo;
import packaccess.PackAccess;
class InheritPack extends PackAccess {
    void display() {
        System.out.println(x);
        show();
    }
}
public class PackAccDemo {
    public static void main(String[] args) {
        InheritPack ob1 = new InheritPack();
        //System.out.println(ob1.x);
        ob1.display();
    }
}
```

Importing Packages - 1

- There are 3 different ways to refer to any class that is present in a different package:
- **Using fully qualified name:** If you use fully qualified name to import any class into your program, then only that particular class of the package will be accessible in your program, other classes in the same package will not be accessible. For this approach, there is no need to use the import statement. But you will have to use the fully qualified name every time you are accessing the class or the interface, which can look a little untidy if the package name is long.
- This is generally used when two packages have classes with same names. For example: java.util and java.sql packages contain Date class.

Importing Packages - 1

```
package pack;
public class A {
    public void msg() {
        System.out.println("Hello");
    }
}
//save as B.java
package mypack;
class B {
    public static void main(String args[]) {
        pack.A obj = new pack.A();  //using fully qualified name
        obj.msg();
    }
}
```

Importing Packages - 2

- **To import only the class/classes you want to use**
  - If you import packagename.classname then only the class with name **classname** in the package with name **packagename** will be available for use.

Importing Packages - 2

```
//save as A.java
package pack;
public class A {
    public void msg() {
        System.out.println("Hello");
    }
}

//save as B.java
package mypack;
import pack.A;
class B {
    public static void main(String args[]) {
        A obj = new A();
        obj.msg();
    }
}
```

Importing Packages - 3

- **To import all the classes from a particular package**If you use packagename.*, then all the classes and interfaces of this package will be accessible but the classes and interface inside the subpackages will not be available for use.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

Importing Packages - 2

```
//save as First.java
package learnjava;
public class First{
   public void msg() {
      System.out.println("Hello");
   }
}

//save as Second.java
package Java;
import learnjava.*;
class Second {
   public static void main(String args[]) {
      First obj = new First();
      obj.msg();
   }
}
```

Some Standard Java Packages

Subpackage	Description
java.lang	Contains a large number of general-purpose classes
java.io	Contains the I/O classes
java.net	Contains classes that support networking
java.applet	Contains classes for creating applets
java.awt	Contains classes that support the Abstract Window Toolkit (AWT)
java.util	Contains various utility classes and the Collections Framework

Static Import (JDK 5 onwards)

- static import is used to import the static members of a class or interface.
  - With static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class.

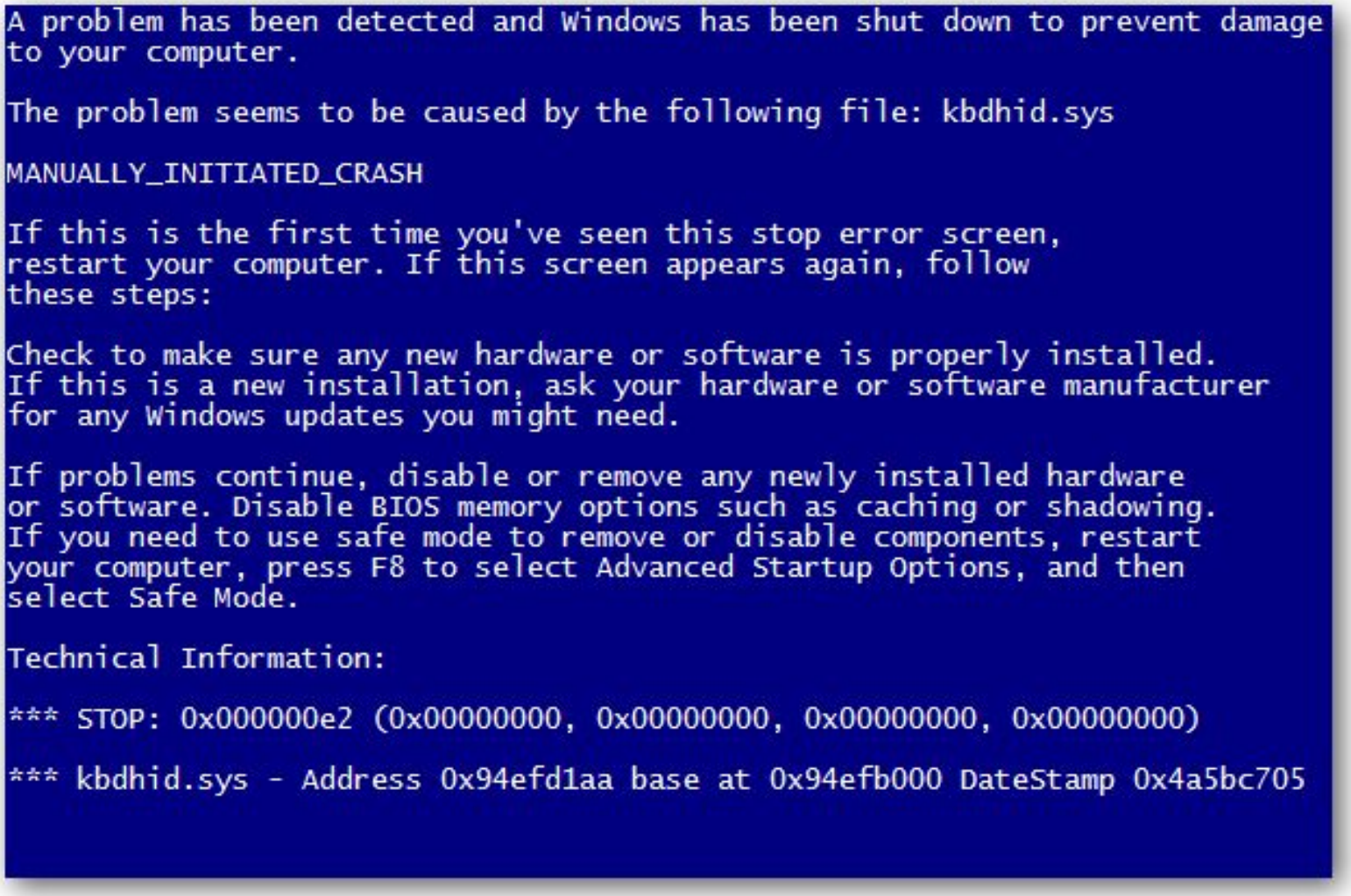
```
package staticimport;
import java.util.Scanner;
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
public class StaticImport {
    public static void main(String[] args) {
        double a, b, c, r1, r2;
        Scanner in = new Scanner(System.in);
        System.out.print("Enter the coefficients: ");
        a = in.nextDouble();
        b = in.nextDouble();
        c = in.nextDouble();
        System.out.println(a + " " + b + " "  + c);
        r1 = (-b + Math.sqrt(pow(b, 2) - 4 * a* c)) / ( 2 * a);
        r2 = (-b - Math.sqrt(pow(b, 2) - 4 * a* c)) / ( 2 * a);
        System.out.println("First root is " + r1);
        System.out.println("Second root is " + r2);
    }
}
```

Additional Points about Packages

- A package is always defined as a separate folder having the same name as the package name.
- Store all the classes in that package folder.
- All classes of the package which we wish to access outside the package must be declared public.
- All classes within the package must have the package statement as its first line.
- All classes of the package must be compiled before use (So that they are error free)

Exception Handling

Runtime Error



Runtime Error

Concepts of Exception Handling

- Exception is an error event that can happen during the execution of a program and disrupts the normal flow.
- Java provides a robust and object oriented way to handle exception scenarios.
- An exception can occur for many different reasons.
  - User has entered an invalid data.
  - A file that needs to be opened is not found.
  - A network connection has been lost in the middle of communications.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may catch the exception and handle the exception.

- Exception
  - Exception is an abnormal condition that arises at run time.
  - Event that disrupts the normal flow of the program.
  - It is an object which is thrown at runtime.
- Exception Handling
  - Exception Handling is a mechanism to handle runtime errors.
  - Normal flow of the application can be maintained.
  - Exception handling done with the exception object thrown at run time.

Types of Errors

- Syntax errors
  - Arise because the rules of the language have not been followed.
  - They are detected by the compiler.
- Runtime errors
  - Occur while the program is running if the environment detects an operation that is impossible to carry out.
- Logic errors
  - Occur when a program doesn't perform the way it was intended to.

Uncaught Exceptions

- What happens when you don't handle exceptions?

```
class Example1
{
   public static void main(String args[])
   {
      int d = 0;
      int a = 42 / d;
   }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.
- This causes the execution of Example1 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.
- Since we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.
- Any exception that is not caught by our program will ultimately be processed by the default handler.

```
java.lang.ArithmeticException: / by zero
at Example1.main(Example1.java:6)
```

Concepts of Exception Handling

- Java exception handling is managed using five keywords:
  - try,
  - catch,
  - throw,
  - throws,
  - finally.

Exception Handling

- Program statements that you want to monitor for exceptions are contained within a try block.
- If an exception occurs within the try block, the exception object is thrown.
- Your code can catch this exception (using catch) and handle it in some rational/appropriate manner.
- System-generated exceptions are automatically thrown by the Java run-time system. 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 Handling

```
try {
// block of code to monitor for errors
// the code you think can raise an exception
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// optional
finally {
// block of code to be executed after try block ends
}
```

Using try and catch

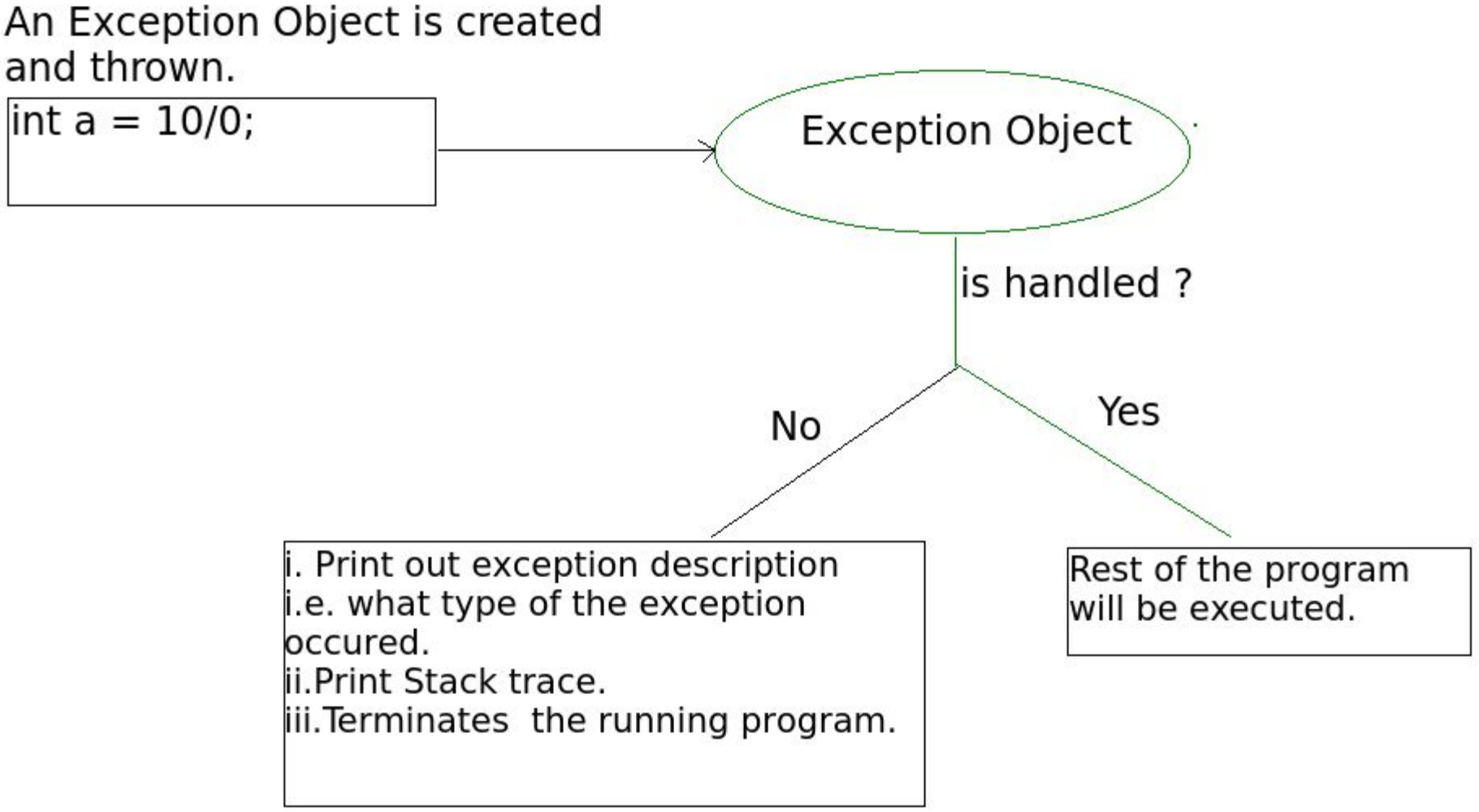
```
class Example1
{
   public static void main(String args[])
   {
      int d, a;
      try
      {
         d = 0;
         a = 42 / d;
         System.out.println("Not reachable");
      }
      catch (ArithmeticException e)
      {
         System.out.println(“Error: Division by zero.");
      }
   }
}
```

Once an exception is thrown, program control transfers out of the try block into the catch block.

Using Multiple catch Clauses

```
public class ExceptionDemo {
    public static void main(String[] args) {
        int [] numer = {4, 8, 16, 32, 64, 128, 256, 512};
        int [] denom = {2, 0, 4, 4, 0, 8};
        for(int i=0; i<numer.length; i++) {
            try {
                double result = numer[i] / denom[i];
                System.out.println(numer[i] + " / " + denom[i] + " = " + result);
            }
            catch(ArithmeticException e) {
                System.out.println("ERROR: Division by zero!");
            }
            catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("No matching element found!");
            }
        }
    }
}
```

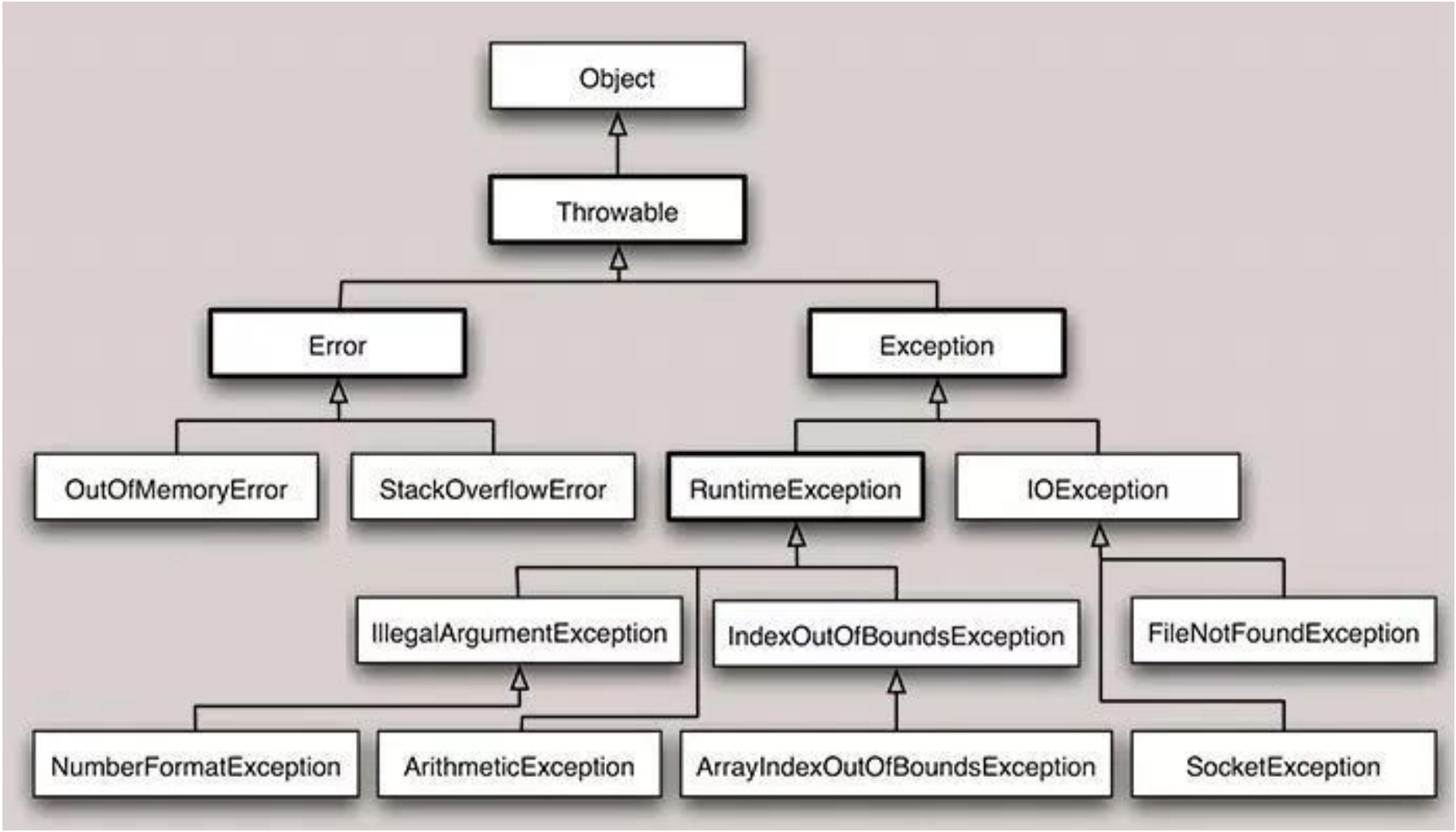
Summary


An Exception Object is created and thrown.
int a = 10/0;
Exception Object
is handled ?
No
Yes
i. Print out exception description i.e. what type of the exception occured.
ii.Print Stack trace.
iii.Terminates the running program.
Rest of the program will be executed.

Exception Hierarchy

- In Java all exceptions are represented by classes.
- All exception classes are derived from a class called Throwable.
  - There are two direct subclasses of Throwable.
    - Error: Exceptions of type Error are related to errors that are beyond your control, such as those that occur in the JVM itself (Out of Memory, Stack Overflow).
    - Exception: Errors that result from program activity are represented by subclasses of Exception.
      - Divide-by-zero, array boundary, I/O errors fall into this category.
      - Our program should handle exceptions of these types.

Exception Hierarchy



Catching Subclass Exceptions

- A catch clause for a superclass will also match any of its subclasses.
  - If you catch exceptions of both a superclass type and a subclass type, put the subclass first in the catch sequence.
    - If you do not, then the superclass catch will also catch all derived classes.
  - This is mandatory in Java because putting the superclass first causes unreachable code to be created, since the subclass catch can never execute.
    - In such cases, a compile-time error occurs.

This code will not compile!

```
public class ExceptionDemo {
   public static void main(String[] args) {
      int [] numer = {4, 8, 16, 32, 64, 128, 256, 512};
      int [] denom = {2, 0, 4, 4, 0, 8};
      for(int i=0; i<numer.length; i++) {
         try {
            double result = numer[i] / denom[i];
            System.out.println(numer[i] + " / " + denom[i] + " = " + result);
         }
         catch(Exception e) {
            System.out.println("No matching element found!");
         }
         catch(ArithmeticException e) {
            System.out.println("ERROR: Division by zero!");
         }
      }
   }
}
```

error: exception ArithmeticException has already been caught

try Blocks can be Nested

```
public class ExceptionDemo {
   public static void main(String[] args) {
      int [] numer = {4, 8, 16, 32, 64, 128, 256, 512};
      int [] denom = {2, 0, 4, 4, 0, 8};
      try { // Outer try
         for(int i=0; i<numer.length; i++) {
            try { // Nested try
               double result = numer[i] / denom[i];
               System.out.println(numer[i] + " / " + denom[i] + " = " + result);
            }
            catch(ArithmeticException e) {
               System.out.println("ERROR: Division by zero!");
            }
         }
      }
      catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("No matching element found!");
            System.out.println("ERRROR: Program terminated.");
      }
   }
}
```

Throwing an Exception

- So far, we have been catching the exceptions thrown by the JVM.
- However, it is possible to manually throw an exception by using the **throw** statement.
- Its general form is:
  - throw *exceptionObj*;
- An exception caught by one catch can be rethrown so that it can be caught by an outer catch.

```
class Rethrow {
   public static void genException() {
      int [] numer = {4, 8, 16, 32, 64, 128, 256, 512};
      int [] denom = {2, 0, 4, 4, 0, 8};
      for(int i=0; i<numer.length; i++) {
         try {
            double result = numer[i] / denom[i];
            System.out.println(numer[i] + " / " + denom[i] + " = " + result);
         }
         catch(ArithmeticException e) {
            System.out.println("ERROR: Division by zero!");
         }
         catch(ArrayIndexOutOfBoundsException  e) {
            throw e;
         }
      }
   }
}
```

Here, divide-by-zero errors are handled locally, by genException(), but an array boundary error is rethrown.

```
public class RethrowException {
    public static void main(String[] args) {
        try {
            Rethrow.genException();
        }
        catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("No matching element found.");
            System.out.println("Fatal Error…Program terminated...");
        }
    }
}
```

The rethrown exception is caught by main()

A Closer Look at Throwable

- When an exception occurs, the catch clause specifies an exception type and a parameter.
- The parameter receives the exception object.
- Since all exceptions are subclasses of Throwable, all exceptions support the methods defined by Throwable.
- Some commonly used methods are listed next…

Commonly Used Methods of Throwable

Method	Description
Throwable fillInStackTrace()	Returns a Throwable object that contains a completed stack trace.
String getLocalizedMessage()	Returns a localized description of the exception.
String getMessage()	Returns a description of the exception.
void printStackTrace()	Displays the stack trace.
void printStackTrace(PrintStream stream)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter stream)	Sends the stack trace to the specified stream.
String toString()	Returns a String object containing a complete description of the exception.

Using finally

- finally block will be executed whenever execution leaves a try/catch block, no matter what condition causes it.
  - i.e., whether try block ends normally or not, the last code executed is that defined by finally.
  - The finally block is also executed if any code within the try block or any of its catch clauses return from a method.

```
class Finally {
   public static void genException(int divisor) {
      int res, nums[] = new int[4];
      System.out.println("Received " + divisor);
      try {
         switch(divisor) {
            case 0:
               res = 12 / divisor; break;
            case 1:
               nums[4] = 4; break;
            case 2: return;
         }
      }
      catch(ArithmeticException e) {
         System.out.println("ERROR: Division by zero!"); return;
      }
      catch(ArrayIndexOutOfBoundsException e) {
         System.out.println("Cannot insert into array");
      }
      finally {
         System.out.println("Leaving try...");
      }
   }
}
```

```
public class FinallyDemo {
    public static void main(String[] args) {
        for(int i=0; i<3; i++){
            Finally.genException(i);
            System.out.println();
        }
    }
}
```

Using throws

- If a method generates an exception that it does not handle, it must declare that exception in a throws clause.
  - ret_type methodName(param_list) throws except_list {
  - // body
  - }
- Here, except_list is a comma-separated list of exceptions that the method might throw outside of itself.
  - If a method has code that may generate unchecked, it need not use throws clause.

```
public class ThrowsDemo {
   static void checkAge(int age) throws SecurityException {
      if (age < 18) {
         throw new SecurityException("Access denied - You are underage!!");
      }
      else {
         System.out.println("Access granted - You are old enough!");
      }
 }
   public static void main(String[] args) {
      try {
            checkAge(21);
      checkAge(15);
      }
      catch(SecurityException e) {
         System.out.println(e.toString());
      }
   }
}
```

Exception Rules

- You cannot have a catch or finally without a try.
- You cannot put code between the try and the catch.
- A try must be followed by either a catch or a finally.

Advantage of Exception Handling

Separating Error-Handling Code from "Regular" Code

```
readFile
{
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Exception Hierarchy

- All exception and errors types are sub classes of class Throwable, which is base class of hierarchy.
  - One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch.
    - NullPointerException is an example of such an exception.
  - Another branch, Error are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE).
    - StackOverflowError is an example of such an error.

Checked Vs. Unchecked Exceptions in Java

- **Checked:** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.
  - For example, consider the Java program that opens file at location "C:\test\a.txt" and prints the first three lines of it.
  - The program doesn't compile, because the function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*.
  - It also uses readLine() and close() methods, and these methods also throw checked exception *IOException*

Checked Exception Example

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

```
Exception in thread "main" java.lang.RuntimeException:
Uncompilable source code - unreported exception
java.io.FileNotFoundException; must be caught or declared to be
thrown at Main.main(Main.java:5)
```

Checked Exception (Fixed...)

```
import java.io.*;

class Main {
    public static void main(String[] args) throws IOException {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

Unchecked Exceptions

- **Unchecked** are the exceptions that are not checked at compiled time.
  - In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception.
  - It is up to the programmers to be civilized, and specify or catch the exceptions.
  - In Java exceptions under Error and RuntimeException classes are unchecked exceptions.

Unchecked Exceptions Example

- The following program compiles fine, but it throws *ArithmeticException*when run. The compiler allows it to compile, because *ArithmeticException* is an unchecked exception.

```
class Example {
   public static void main(String args[]) {
      int x = 0;
      int y = 10;
      int z = y/x;
   }
}
```

```
Exception in thread "main" java.lang.ArithmeticException:
/ by zero at Main.main(Main.java:5)
```

- Why doesn't the complier care about the runtime exceptions?
  - Most Runtime Exceptions come from a problem in your code logic, rather than a condition that fails at runtime in ways that you cannot predict or prevent.
  - A try/catch is for handling exceptional situations, not flaws in your code.
    - Make sure your code doesn't index off the end of an array!
    - Make sure that you don’t perform illegal division!!

Creating Exception Subclasses

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- You can create your own exception sub class simply by extending java **Exception** class.
- The Exception class does not define any methods of its own.
- You can define a constructor for your Exception sub class (not compulsory) and you can override the **toString()** method (inherited from Throwable class) to display your customized message on catch.
- The **getMessage()** method can also be used to display the message regarding the exception.

```
class UnderAgeException extends Exception {
   UnderAgeException(String s) {
      super(s);
   }
   @Override
   public String toString() {
      return "You are not old enough for this!";
   }
}
public class CustomExceptionDemo {
   static void fun(int age) {
      try {
         if(age < 18)
              throw new UnderAgeException("InvalidAgeException");
              System.out.println("Welcome! Have great fun!!");
      }
      catch(UnderAgeException e) {
              System.out.println(e);
              System.out.println(e.getMessage());
              e.printStackTrace();
      }
   }
```

```
   public static void main(String[] args) {
         fun(22);
         fun(15);
     }
   }
```

String Handling

String Fundamentals

- In Java, a string is a sequence of characters.
  - But instead of character arrays, strings in Java are implemented as objects of String class.
- String class implements the following interfaces: Comparable<String>, CharSequence, and Serializable
  - The Comparable interface specifies how objects are compared.
  - The CharSequence interface defines a set of methods that are applicable to a character sequence.
  - The Serializable interface indicates that the state of a String can be saved and restored using Java's serialization mechanism.
- Strings are immutable.
  - You cannot change the characters that comprise that string.
  - However, you can create a new string which is an altered version of an existing string.
  - Also, a variable declared as a String reference can be changed to refer to a different String object.

The String Constructors

- Many overloaded constructors are available in String class.
  - String(char [] chr)
  - String(char [] che, int startIndex, int numChars)
  - String(String strObj)
  - String(byte [] chr)
  - String(byte [] che, int startIndex, int numChars)

String-related Language Features

1. String Literals
   - We can use a string literal to initialize a String object.
     - String str = “Hello World”;
     - We can pass a string literal as an argument to a method that is expecting a String.
2. String concatenation
   - We can use the + operator to concatenate two strings and produce a new String object.

String-related Language Features

3. String concatenation with other data types
   - int age = 19;
   - String s1 = “He is “ + age + “ years old”;
   - Compiler will convert an operand to its string equivalent whenever the other operand of the + is an instance of String.
   - String s2 = “Four:” + 2 + 2 // ???
   - String s2 = “Four:” + (2 + 2) // ???

Overriding toString()

- When Java converts an object into its string representation, it does so by calling the toString() method (defined by Object class).
  - toString() returns a string that described the object.

```
class Box {
    int wid, hgt, dep;
    Box(int w, int h, int d) {
        wid = w;
        hgt = h;
        dep = d;
    }
    String toString() {
        return “Dimensions are “ + wid + “ by “ + hgt + “ by “ + dep”;
    }
}
```

The length() method

- Since strings are not arrays, they don't have length as field; instead they have a method called length() that returns the length of the string.
  - A string's length is the number of characters that it contains.
  - String s = "Hello World";
  - System.out.println(s.length); // prints 11

Obtaining characters within a String

- String class provides three ways in which characters can be obtained from a String object.

1. char charAt(int *where*)
   - returns the character at the specified index named *where*
   - The value of *where* must be nonnegative and specify a location in the string.
   - Example:
     - String s = “GIT, Belgaum”;
     - System.out.println(s.charAt(5)); // Prints B

Obtaining characters within a String

2. void getChars(int *sourceStart*, int *sourceEnd*, char [] *target*, int *targetStart*)
   - *sourceStart*: index of the beginning of the substring to obtain
   - *sourceEnd*: index that is one past the end of the desired substring
   - Thus the substring contains the characters from *sourceStart* through *sourceEnd* - 1.
   - The array that receives the substring is specified by *target*.
   - targetStart: index within target at which the substring will be copied.
   - Example:
     - String s = “Welcome to my world!”;
     - int start = 11, end = 19;
     - char [] arr = new char[end-start];
     - arr.getChars(start, end, arr, 0); // arr will contain “my world”

Obtaining characters within a String

3. char [] toCharArray()
   - Converts all the characters in a String object into a character array.
   - Example:
   - String s = “GIT, Belgaum”;
   - char [] arr = s.toCharArray();
   - System.out.println(arr); // Prints GIT, Belgaum

String Comparison

- There are a number of methods that compare strings or substrings within strings.
  - equals() and equalsIgnoreCase()
  - equals() Versus ==
  - regionMatches()
  - startsWith() and endsWith()
  - compareTo() and compareToIgnoreCase()

equals() and equalsIgnoreCase()

- boolean equals(Object str)
- boolean equalsIgnoreCase(Object str)
  - str is the object being compared with the invoking String object.
  - Returns true if the two strings are same; false otherwise.
  - In the first method, comparison is case sensitive.
  - In second method, comparison is case-insensitive.

```
public class EqualsExample{
	public static void main(String args[]){
		String s1="hello";
		String s2="hello";
		String s3="hi";
		String s4 = “HELLO”;
		System.out.println(s1.equals (s2));   // returns true
		System.out.println(s1.equals(s3));   // returns false
		System.out.println(s1.equalsIgnoreCase(s4));   // returns true
	}
}
```

equals Versus ==

- The equals() method compared the characters inside a String object whereas the == operator compares two object references to see whether they refer to the same instance.
  - String s1 = “hello”;
  - String s2 = new String(“hello”);
  - String s3 = s1;
  - System.out.println(s1.equals(s2)); // Prints true
  - System.out.println(s1 == s2)); // Prints false
  - System.out.println(s1 == s3)); // Prints true

regionMatches()

- Compares a subset (that is, a region) of a string with a subset of another string.
  - boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars)
  - boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)
    - startIndex specifies the index at which the region to compare begins within the invoking String object.
    - The string being compared is specified by str2.
    - str2StartIndex specifies the index at which the comparison will start within str2.
    - numChars specifies the length of the region being compared.
    - If ignoreCase is true, the case of the characters is ignored.

startsWith() and endsWith()

- boolean startsWith(String str)
  - Determines whether a given **String** begins with a specified string.
- boolean endsWith(String str)
  - Determines whether the **String** in question ends with a specified string.
- Example
  - String str = “Gogte Institute of Technology”;
  - str.startsWith(“Gogte”); // true
  - str.endsWith(“ology”); // true
- boolean startsWith(String str, int startIndex)
  - Here startIndex specifies the index into the invoking string at which point the search will begin.
  - str.startsWith(“gte”, 2); // true

compareTo() and compareToIgnoreCase()

- int compareTo(String str)
- int compareToIgnoreCase(String str)
  - Perform lexicographic comparison
  - A string is less than another if it comes before the other in dictionary order.
  - String s1 = "Belgium";
  - System.out.println(s1.compareTo("Belgaum")); // 8
  - System.out.println(s1.compareTo("belgaum")); // -32
  - System.out.println(s1.compareTo(“Belgaum")); // 0
  - System.out.println(s1.compareToIgnoreCase("belgaum")); // 0

Using indexOf() and lastIndexOf()

- int indexOf(str)
  - Searches the invoking string for the substring specified by str. Returns the index of the first match or -1 on failure
- int lastIndexOf(str)
  - Searches the invoking string for the substring specified by str. Returns the index of the last match or -1 on failure
  - String s1="this is idea of example";
  - int index1=s1.indexOf("is");//returns the index of ‘is’
  - int index2=s1.indexOf("idea");//returns the index of ‘idea’
  - System.out.println(index1 + " " + index2); //2 8
  - String str = “Welcome to home sweet home";
  - int index = str.lastIndexOf(“home");
  - System.out.println(index); // 22

Obtaining a Modified String

- String substring(int startIndex)
  - Returns a copy of the substring that begins at *startIndex* and includes the remainder of the invoking string.
- String substring(int startIndex, int endIndex)
  - Returns a copy of the string containing all the characters from *startIndex*, up to, but not including, *endIndex.*
- String replace(char original, char replacement)
  - Character specified by *original* is replaced by the character specified by *replacement* and the modified copy of the invoking string is returned.
- String replace(CharSequence original, CharSequence replacement)
  - The sequences can be objects of type String as String implements CharSequence interface.
- String trim()
  - Deletes leading and trailing whitespaces (typically spaces, tabs and newline characters) from a string and returns a new string.

- String str= new String("quick brown fox jumps over the lazy dog");
- System.out.println("Substring starting from index 15:");
- System.out.println(str.substring(15));
- System.out.println("Substring from index 15 and ending at 20:");
- System.out.println(str.substring(15, 20));

```
Substring starting from index 15:
 jumps over the lazy dog
Substring starting from index 15 and ending at 20:
 jump
```

- String str = new String("Good morning. Have a good day");
- System.out.println(str.replace('o', '$'));
  - // G$$d m$rning. Have a g$$d day

- String s1="My name is Bond. James Bond!";
- String replaceString=s1.replace("Bond","Madison");
- System.out.println(replaceString);
  - // My name is Madison. James Madison!

- String str1 = new String("   How are you??   ");
- String str2 = str.trim());

Changing the case of Characters in a String

- String toLowerCase()
- String toUpperCase()
  - Both methods return a String that contains the uppercase or lowercase equivalent of the invoking String.
- Example:
  - String str1 = new String("This is a test.");
  - String str2 = str1.toLowerCase();
  - String str3 = str1.toUpperCase();
  - System.out.println(str2); // this is a test.
  - System.out.println(str3); // THIS IS A TEST.

StringBuffer and StringBuilder

- Both offer capabilities similar to String with one important addition:
  - They contain strings that can be modified (mutable).
  - We can insert characters and substrings in the middle or append to the end.
  - We can also delete characters.
  - Both StringBuffer and StringBuilder will automatically grow or shrink as a result of such operations.
  - Operations such as setCharAt(), append(), insert() and delete() are supported.

Reverse a string

```
import java.util.Scanner;
class ReverseString {
	public static void main(String args[]) {
		String str,revStr = "";
		Scanner in = new Scanner(System.in);
		System.out.print("Enter a string :");
		str= in.nextLine();
		int len= str.length();
			for ( int i = len- 1 ; i >= 0 ; i-- )
				revStr= revStr+ str.charAt(i);
			System.out.println("Reverse String is: "+revStr);
	}
}
```

Palindrome

```
import java.util.Scanner;
public class Palindrome {
   public static void main(String[] args) {
            String str, revStr="";
            Scanner in = new Scanner(System.in);
      System.out.print("Enter a string :");
      str= in.nextLine();
      int len= str.length();
            for ( int i = len- 1 ; i >= 0 ; i-- )
                  revStr= revStr + str.charAt(i);
      if(str.equals(revStr))
                  System.out.println("Entered string is a palindrome");
            else
                  System.out.println("Entered string is not a palindrome");
   }
}
```

- Two strings will be anagram to each other if and only if they contain the same number of characters (order of the characters doesn't matter).
- That is, If the two strings are anagram to each other, then one string can be rearranged to form the other string. For Example:
  - creative and reactive are anagrams.
- Write a Java program to test whether two strings are anagrams or not.

```
static void isAnagram(String str1, String str2) {
      String s1 = str1.replaceAll("\\s", "");  // Remove whitespace character
      String s2 = str2.replaceAll("\\s", "");  // Remove whitespace character
      boolean status = true;
      if (s1.length() != s2.length()) {
         status = false;
      }
      else {
         char[] ArrayS1 = s1.toLowerCase().toCharArray();
         char[] ArrayS2 = s2.toLowerCase().toCharArray();
         Arrays.sort(ArrayS1);
         Arrays.sort(ArrayS2);
         status = Arrays.equals(ArrayS1, ArrayS2);
      }
      if (status) {
         System.out.println(s1 + " and " + s2 + " are anagrams");
      }
      else {
         System.out.println(s1 + " and " + s2 + " are not anagrams");
      }
   }
```

- Given a string and a word, count the number of the occurrence of the given word in the string and print the number of occurrence of the word.

```
static int countOccurences(String str, String word) {
    // split the string by spaces
    String a[] = str.split(" ");

    // search for pattern
    int count = 0;
    for (int i = 0; i < a.length; i++)  {
      // if match found increase count
      if (word.equals(a[i]))
                count++;
    }
    return count;
}
```

- Given a string and a substring, the task is to replace all occurrences of the substring with space. We also need to remove trailing and leading spaces created due to this.
  - Input: str = “LIELIEILIEAMLIECOOL”, sub = “LIE”
  - Output: I AM COOL

```
static String extractSecretMessage(String Str, String Sub) {
    // Replacing all occurrences of Sub in Str by empty spaces
    Str = Str.replaceAll(Sub, " ");
    // Removing unwanted spaces in the start and end
    Str = Str.trim();
    return Str;
  }
```

- Given a string, Write a program to remove all the occurrences of a character in the string.

```
static void removeChar(String s, char c) {
    int j, count = 0, n = s.length();
    char []t = s.toCharArray();
    for (int i = j = 0; i < n; i++) {
        if (t[i] != c)
        t[j++] = t[i];
        else
            count++;
    }
    //In Java, char arrays and strings are not null terminated
    //Remove unwanted characters
    while(count > 0) {
        t[j++] = '\0';
        count--;
    }
    System.out.println(t);
}
```

End of Unit - 4