RUNGTA

Exception Handling

Unit - III

© RCET, Bhilai

RUNGTA

Syntax Errors, Runtime Errors, and Logic Errors

there are three categories of errors: syntax errors, runtime errors, and logic 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.

© RCET, Bhilai

Runtime Errors

```
 1  import java.util.Scanner;
 2
 3  public class ExceptionDemo {
 4    public static void main(String[] args) {
 5      Scanner scanner = new Scanner(System.in);
 6      System.out.print("Enter an integer: ");
 7      int number = scanner.nextInt();
 8
 9      // Display the result
10      System.out.println(
11        "The number entered is " + number);
12    }
13  }
```

If an exception occurs on this line, the rest of the lines in the method are skipped and the program is terminated.

Terminated.

Run

© RCET, Bhilai

Catch Runtime Errors

RUNGTA

Run

```
1  import java.util.*;
2
3  public class HandleExceptionDemo {
4    public static void main(String[] args) {
5      Scanner scanner = new Scanner(System.in);
6      boolean continueInput = true;
7
8      do {
9        try {
10         System.out.print("Enter an integer: ");
11         int number = scanner.nextInt();
12
13         // Display the result
14         System.out.println(
15           "The number entered is " + number);
16
17         continueInput = false;
18       }
19       catch (InputMismatchException ex) {
20         System.out.println("Try again. (" +
21           "Incorrect input: an integer is required)");
22         scanner.nextLine(); // discard input
23       }
24     } while (continueInput);
25   }
```

If an exception occurs on this line, the rest of lines in the try block are skipped and the control is transferred to the catch block.

© RCET, Bhilai

RUNGTA

What is an Exception?

- An Exception is abnormal situation (or) unexpected situation in the normal flow of the program execution.
- Because of Exceptions the flow of program execution is getting disturbed so that program execution may continue (or) may not be continued.
- When ever the exception is occurred, handling those exceptions called as “Exception Handling”.

© RCET, Bhilai

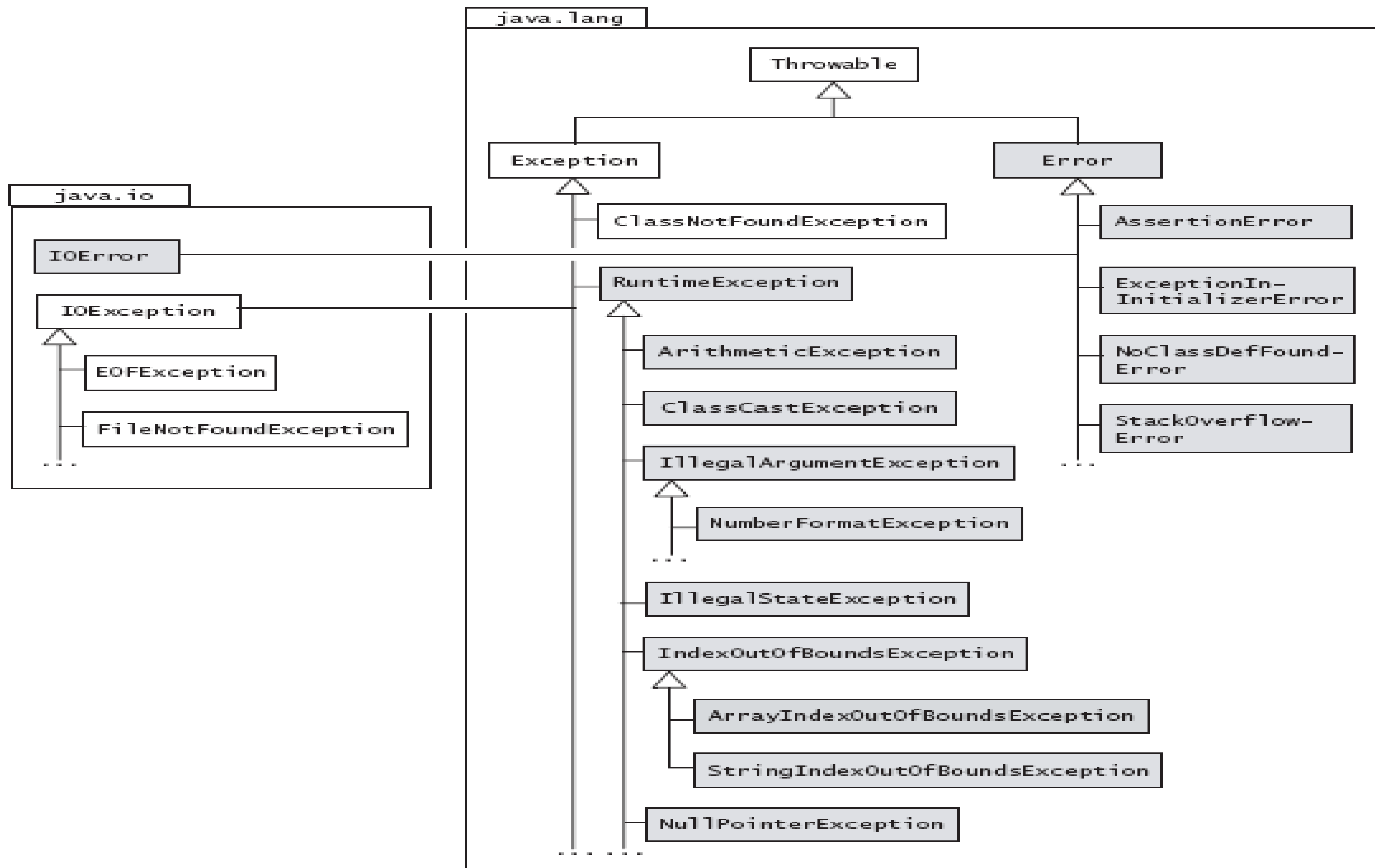
RUNGTA

Advantages by handling Exceptions

- It allows you to fix the error.
- It prevents the program from automatically termination.
- Default exception handler provided by the Java run-time system is useful for debugging.

© RCET, Bhilai

Partial Exception Inheritance Hierarchy



© RCET, Bhilai

The Exception Class

- The class **Exception** represents exceptions that a program would normally want to catch.
- Its subclass RuntimeException represents many common programming errors that can manifest at runtime.
- Other subclasses of the Exception class define other categories of exceptions, e.g., I/O-related exceptions in the java.io package (IOException, FileNotFoundException, EOFException, IOError).

© RCET, Bhilai

ClassNotFoundException

- The subclass **ClassNotFoundException** signals that the JVM tried to load a class by its string name, but the class could not be found.
- A typical example of this situation is when the class name is misspelled while starting program execution with the java command.
- The source in this case is the JVM throwing the exception to signal that the class cannot be found and therefore execution cannot be started.

© RCET, Bhilai

RUNGTA

The RuntimeException Class

- Runtime exceptions are all subclasses of the java.lang.RuntimeException class, which is a subclass of the Exception class.
- As these runtime exceptions are usually caused by program bugs that should not occur in the first place, it is usually more appropriate to treat them as faults in the program design and let them be handled by the default exception handler.

© RCET, Bhilai

RUNGTA

ArithmeticException

- This exception represents situations where an illegal arithmetic operation is attempted, e.g., integer division by 0. It is typically thrown by the JVM.

© RCET, Bhilai

ArrayIndexOutOfBoundsException

- Java provides runtime checking of the array index value, i.e., out-of-bounds array indices.
- The subclass **ArrayIndexOutOfBoundsException** represents exceptions thrown by the JVM that signal out-of-bound errors specifically for arrays, i.e., an invalid index is used to access an element in the array. The index value must satisfy the relation 0 <= *index value* < *length of the array.*

© RCET, Bhilai

RUNGTA

ClassCastException

- This exception is thrown by the JVM to signal that an attempt was made to cast a reference value to a type that was not legal, e.g., casting the reference value of an Integer object to the Long type.

© RCET, Bhilai

NullPointerException

- This exception is typically thrown by the JVM when an attempt is made to use the null value as a reference value to refer to an object. This might involve calling an instance method using a reference that has the null value, or accessing a field using a reference that has the null value.

© RCET, Bhilai

The Error Class

- The class Error and its subclasses define errors that are invariably never explicitly caught and are usually irrecoverable. Not surprisingly, most such errors are signalled by the JVM. Apart from the subclasses mentioned below, other subclasses of the java.lang.Error class define errors that indicate class linkage (LinkageError), thread (ThreadDeath), and virtual machine (VirtualMachineError) problems.

© RCET, Bhilai

RUNGTA

Checked Exceptions vs. Unchecked Exceptions

❖RuntimeException, Error and their subclasses are known as ***unchecked exceptions*.**

❖All other exceptions are known as ***checked exceptions***, meaning that the compiler forces the programmer to check and deal with the exceptions.

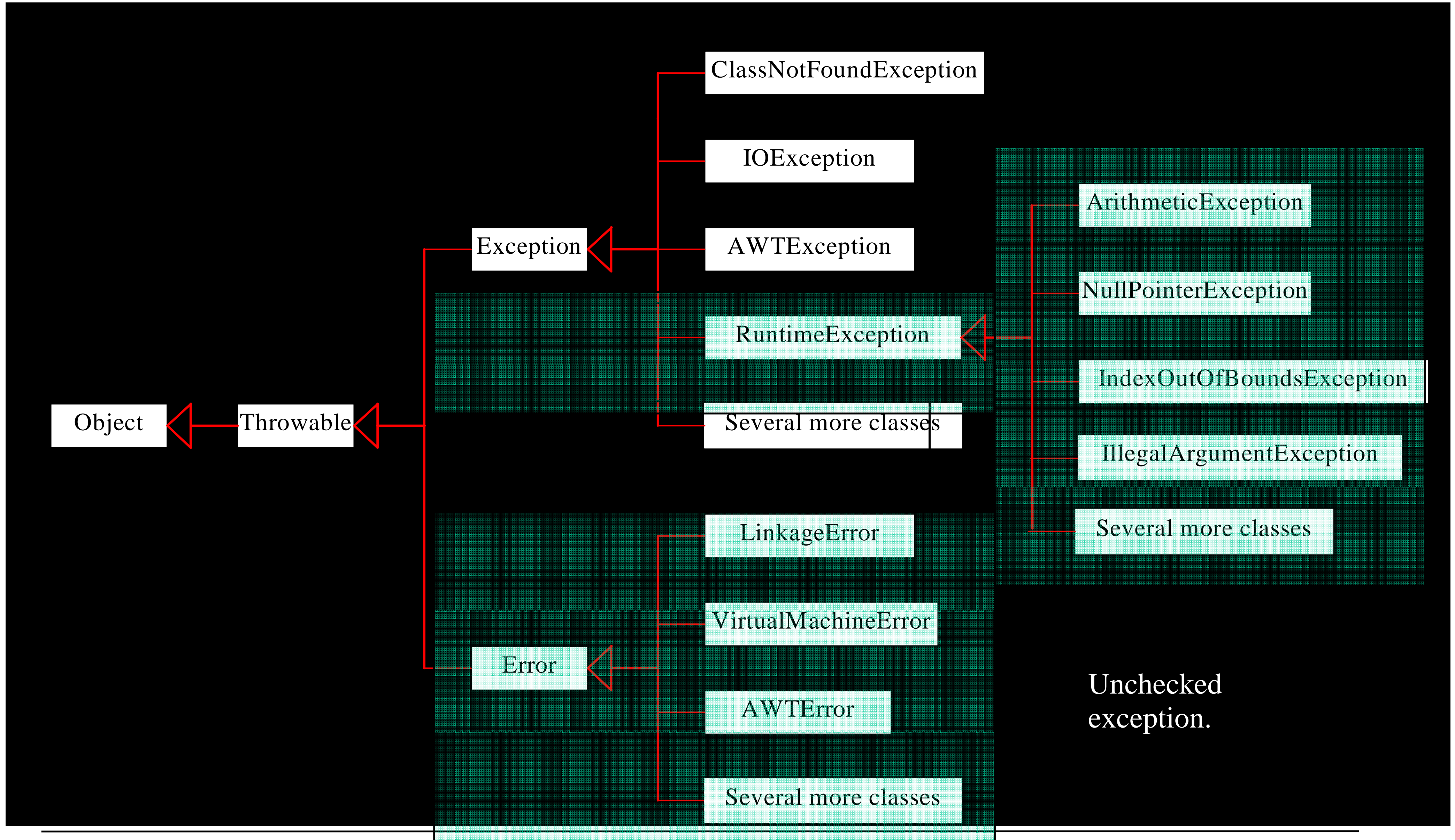
© RCET, Bhilai

Unchecked Exceptions

In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example, a **NullPointerException** is thrown if you access an object through a reference variable before an object is assigned to it; an **IndexOutOfBoundsException** is thrown if you access an element in an array outside the bounds of the array. These are the logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in the program. To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

© RCET, Bhilai

Checked or Unchecked Exceptions



© RCET, Bhilai

Exception Handling

- In java Exception handling mechanism is depending on the following keywords: try, catch, finally, throw, throws
- In java language all these exceptions are handled by using a class called as “Exception”. Which is part of “java.lang” package.

© RCET, Bhilai

RUNGTA

Exception Handling: try, catch, and finally

- The mechanism for handling exceptions is embedded in the try-catch-finally construct,
- which has the following general form:

try { // try block

<statements>

} catch (*<exception type1> <parameter1>) { // catch block*

<statements>

}

...

catch (*<exception typen> <parametern>) { // catch block*

<statements>

} finally { // finally block

<statements>

}

© RCET, Bhilai

RUNGTA

The try Block

- The try block establishes a context for exception handling. Termination of a try block occurs as a result of encountering an exception, or from successful execution of the code in the try block.

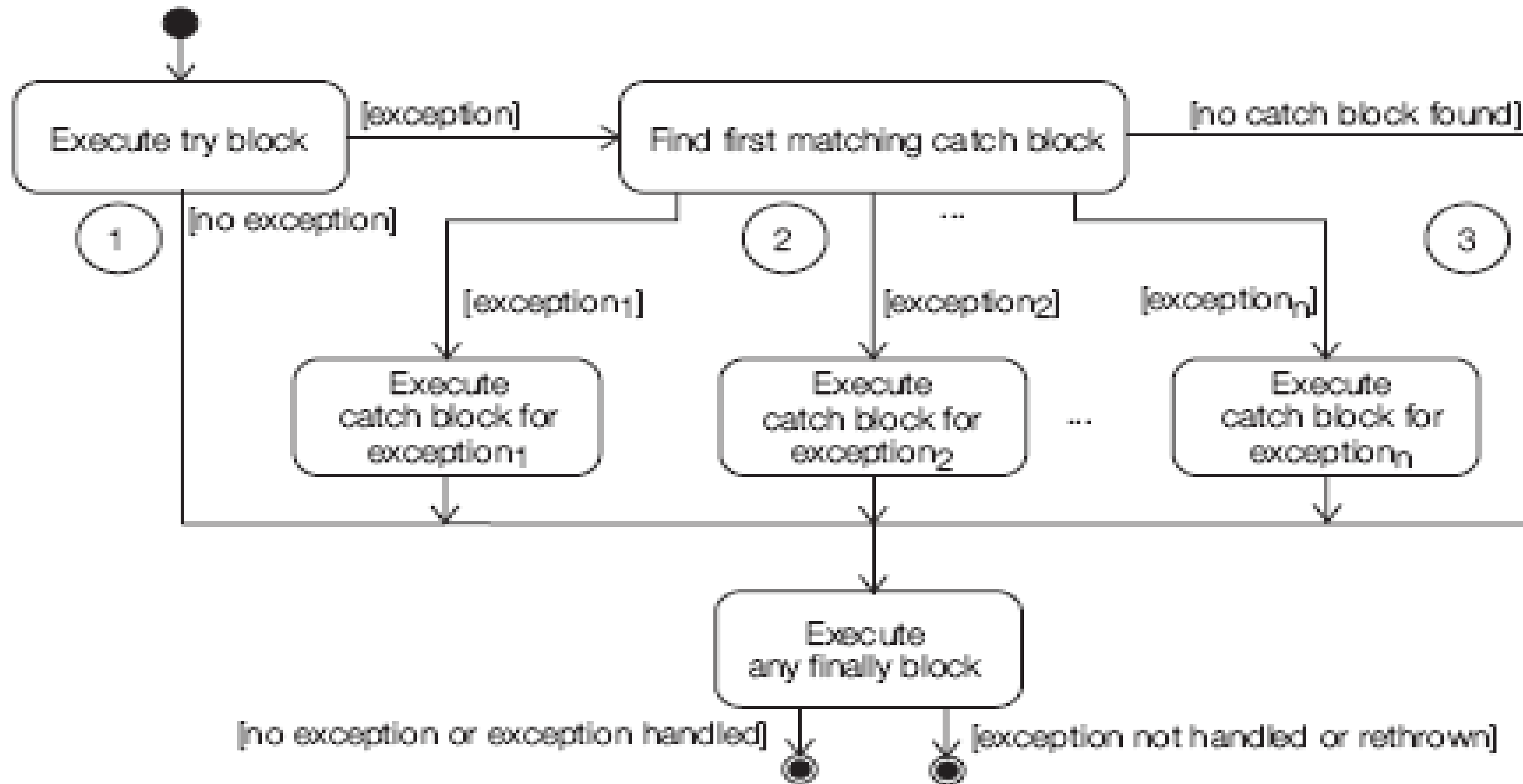
- The catch blocks are skipped for all normal exits from the try block where no exceptions were raised, and control is transferred to the finally block if one is specified

- For all exits from the try block resulting from exceptions, control is transferred to the catch blocks—if any such blocks are specified—to find a matching catch block

© RCET, Bhilai

RUNGTA

The try-catch-finally Construct



© RCET, Bhilai

RUNGTA

The catch Block

- Only an exit from a try block resulting from an exception can transfer control to a
- catch block. A catch block can only catch the thrown exception if the exception is assignable to the parameter in the catch block.
- The code of the first such catch block is executed and all other catch blocks are ignored.
- On exit from a catch block, normal execution continues unless there is any pending exception that has been thrown and not handled. If this is the case, the method is aborted and the exception is propagated up the runtime stack as explained earlier.
- After a catch block has been executed, control is always transferred to the finally block if one is specified. This is always true as long as there is a finally block, regardless of whether the catch block itself throws an exception.

© RCET, Bhilai

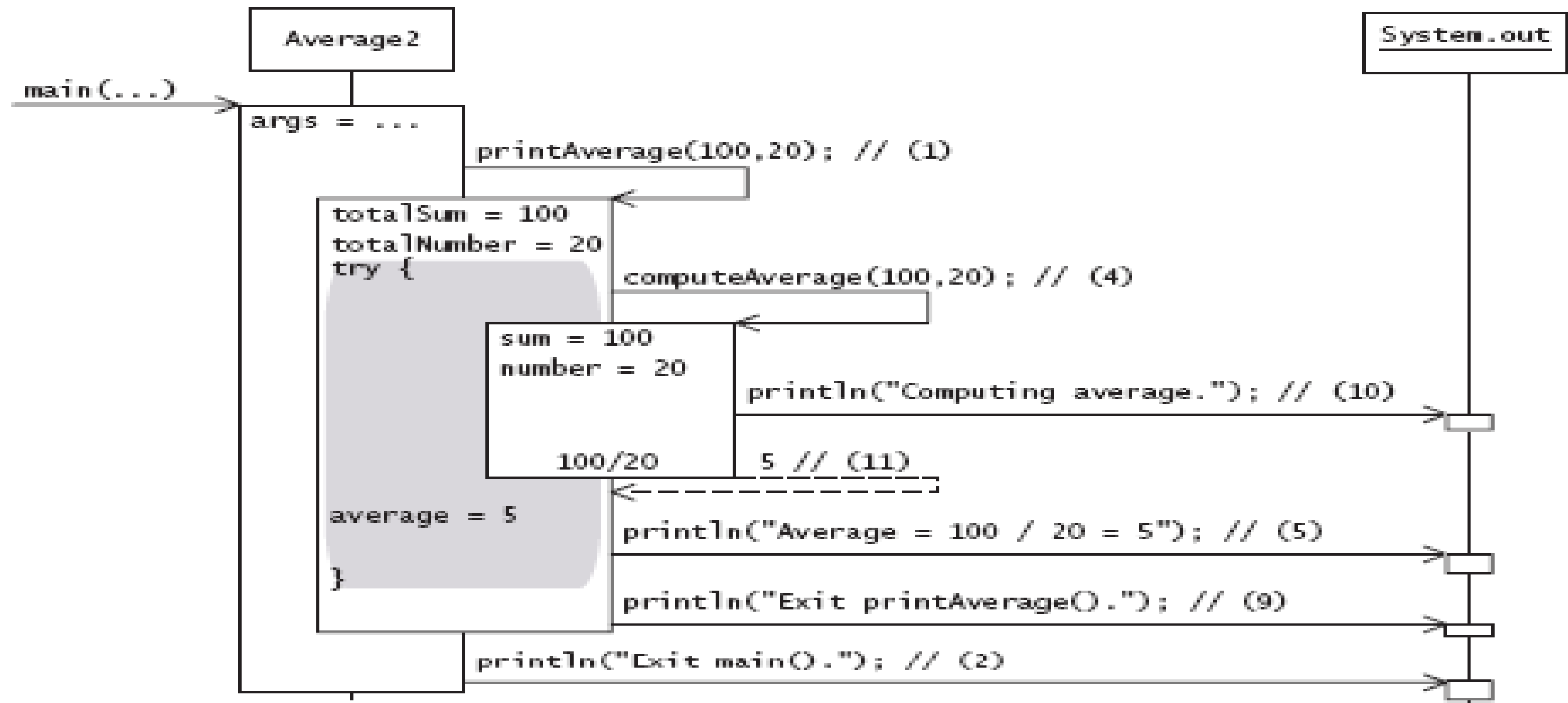
Example of try and catch

```
public class Average2 {

  public static void main(String[] args) {
    printAverage(100, 20);                                    // (1)
    System.out.println("Exit main().");                       // (2)
  }

  public static void printAverage(int totalSum, int totalNumber) {
    try {                                                     // (3)
      int average = computeAverage(totalSum, totalNumber);// (4)
      System.out.println("Average = " +                       // (5)
          totalSum + " / " + totalNumber + " = " + average);
    } catch (ArithmeticException ae) {                        // (6)
      ae.printStackTrace();                                   // (7)
      System.out.println("Exception handled in " +
                         "printAverage().");                  // (8)
    }
    System.out.println("Exit printAverage().");               // (9)
  }

  public static int computeAverage(int sum, int number) {
    System.out.println("Computing average.");                 // (10)
    return sum/number;                                        // (11)
  }
}
```

Run

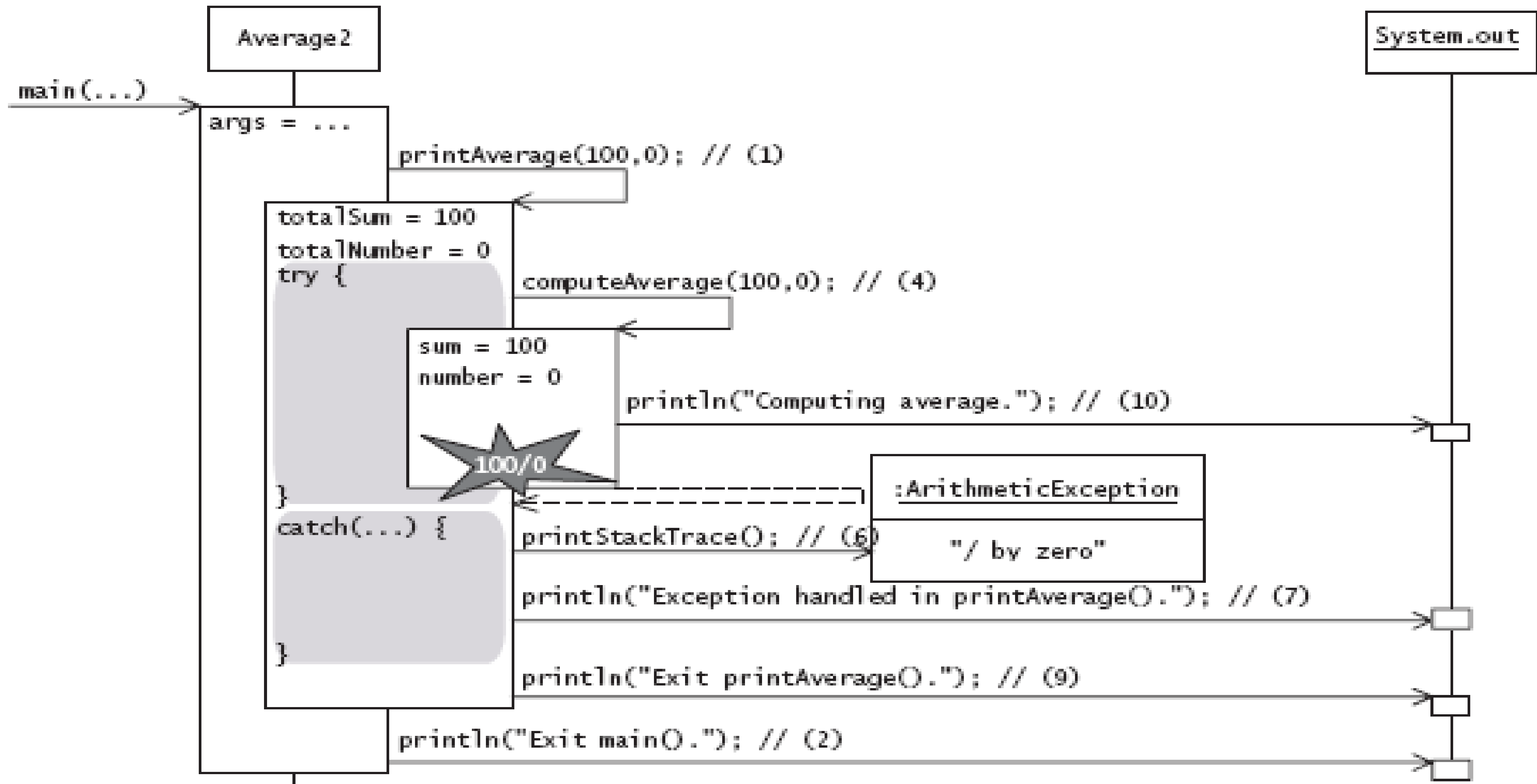
© RCET, Bhilai

Exception Handling (Scenario 1)



© RCET, Bhilai

Exception Handling (Scenario 2)



© RCET, Bhilai

The finally Block

- If the try block is executed, then the finally block is guaranteed to be executed, regardless of whether any catch block was executed.
- Since the finally block is always executed before control transfers to its final destination, the finally block can be used to specify any clean-up code.
- It is also possible to use finally block without catch block.

```
int sum = -1;
try {
sum = sumNumbers();
// other actions
} finally {
if (sum >= 0) calculateAverage();
}
```


© RCET, Bhilai

RUNGTA

Nested try Statements

- The try statement can be nested. That is, a try statement can be inside the block of another try.
- Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.
- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
- If no catch statement matches, then the Java run-time system will handle the exception.

© RCET, Bhilai

Example of Nested Try

```
class NestTry
{
  public static void main(String args[])
  {
        try
       {
          int a = args.length;
          int b = 42 / a;
          System.out.println("a = " + a);
          try {
                 if(a==1)
                  a = a/(a-a);
                 if(a==2)
                 {
                    int c[] = { 1 };
                       c[42] = 99;
                 }   }

        catch(ArrayIndexOutOfBoundsException e)
              {
              System.out.println("Array index out-of-bounds: " + e);
              }    }
   catch(ArithmeticException e)
       {
          System.out.println("Divide by 0: " + e);
       }    }  }
```


© RCET, Bhilai

The throw Statement

- The catching exceptions that are thrown by the Java run-time system.
- it is also possible for a program to throw an exception explicitly.
- By using **throw** keyword we can able to create the exception objects explicitly.
- The general form of throw statement is shown here:
  - **throw <** ***throwableInstance* >;**
  - Here *throwableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.

© RCET, Bhilai

RUNGTA

Example of throw statement

```
class ThrowDemo {
static void demoproc() {
try {
    throw new NullPointerException("demo");
} catch(NullPointerException e) {
    System.out.println("Caught inside demoproc.");
    throw e; // rethrow the exception
} }
public static void main(String args[]) {
try {
    demoproc();
} catch(NullPointerException e) {
    System.out.println("Recaught: " + e);
} } }
```


© RCET, Bhilai

RUNGTA

The throws Clause

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- It can be used by including a **throws clause in the method's declaration.**
- **A throws clause** lists the types of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type **Error or RuntimeException, or any of their subclasses.**

© RCET, Bhilai

The throws Clause…

type method-name(parameter-list) throws <ExceptionType1>, <ExceptionType1>…

```
{
    // body of method
}
```


© RCET, Bhilai

RUNGTA

Advantage of throws Clause

- The main advantage of throws is

1. Escape the Exception because it is a Weaker Exception Handler.

2. throws gives a warning message to the user, who is interested to handle the Exception.

© RCET, Bhilai

RUNGTA

Example of throws Clause

```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
        throwOne();
} catch (IllegalAccessException e) {
        System.out.println("Caught " + e);
} } }
```


© RCET, Bhilai

Review Question

Q.1 Which digits, and in what order, will be printed when the following program is run?

```
public class MyClass {
public static void main(String[] args) {
int k=0;
try {
int i = 5/k;
} catch (ArithmeticException e) {
System.out.println("1");
} catch (RuntimeException e) {
System.out.println("2");
return;
} catch (Exception e) {
System.out.println("3");
} finally {
System.out.println("4");
}
System.out.println("5");
}}
```

Answer: (d)
The program will only print 1, 4, and 5, in that order. The expression 5/k will throw an ArithmeticException, since k equals 0. Control is transferred to the first catch block

Select the one correct answer.
(a) The program will only print 5.
(b) The program will only print 1 and 4, in that order.
(c) The program will only print 1, 2, and 4, in that order.
(d) The program will only print 1, 4, and 5, in that order.
(e) The program will only print 1, 2, 4, and 5, in that order.
(f) The program will only print 3 and 5, in that order.

© RCET, Bhilai

RUNGTA

Review Question

Q.2 Given the following program, which statements are true?

```
public class Exceptions {
public static void main(String[] args) {
try {
if (args.length == 0) return;
System.out.println(args[0]);
} finally {
System.out.println("The end");
} } }
```

Select the two correct answers.

(a) If run with no arguments, the program will produce no output.

(b) If run with no arguments, the program will print "The end".

(c) The program will throw an ArrayIndexOutOfBoundsException.

(d) If run with one argument, the program will simply print the given argument.

(e) If run with one argument, the program will print the given argument followed by "The end".

Answer: (b) and (e)

The finally block will always be executed, no matter how control leaves the try block.

© RCET, Bhilai

RUNGTA

Review Question

Q.3 What will be the result of attempting to compile and run the following program?

Answer: *(d)*

```
public class MyClass {
public static void main(String[] args) {
RuntimeException re = null;
throw re;
} }
```

Select the one correct answer.

(a) The code will fail to compile because the main() method does not declare that it throws RuntimeException in its declaration.

(b) The program will fail to compile because it cannot throw re.

(c) The program will compile without error and will throw java.lang.Runtime-Exception when run.

(d) The program will compile without error and will throw java.lang.NullPointerException when run.

(e) The program will compile without error and will run and terminate without any output.

© RCET, Bhilai

RUNGTA

Review Question

Q.4Which statements are true?

Select the two correct answers.

(a) If an exception is not caught in a method, the method will terminate and normal execution will resume.

(b) An overriding method must declare that it throws the same exception classes as the method it overrides.

(c) The main() method of a program can declare that it throws checked exceptions.

(d) A method declaring that it throws a certain exception class may throw instances of any subclass of that exception class.

(e) finally blocks are executed if, and only if, an exception gets thrown while inside the corresponding try block.

Answer: *(c) and (d)*

© RCET, Bhilai

RUNGTA

Review Question

Q.5 Which digits, and in what order, will be printed when the following program is compiled and run?

```
public class MyClass {
public static void main(String[] args) {
try {
f();
} catch (InterruptedException e) {
System.out.println("1");
throw new RuntimeException();
} catch (RuntimeException e) {
System.out.println("2");
return;
} catch (Exception e) {
System.out.println("3");
} finally {
System.out.println("4");
}
System.out.println("5");
}
// InterruptedException is a direct subclass of Exception.
static void f() throws InterruptedException {
throw new InterruptedException("Time for lunch.");
} }
```

Select the one correct answer.

(a) The program will print 5.
(b) The program will print 1 and 4, in that order.
(c) The program will print 1, 2, and 4, in that order.
(d) The program will print 1, 4, and 5, in that order.
(e) The program will print 1, 2, 4, and 5, in that order.
(f) The program will print 3 and 5, in that order.

Answer: (b) An InterruptedException is handled in the first catch block. Inside this block a new RuntimeException is thrown. This exception was not thrown inside the try block and will not be handled by the catch blocks, but will be sent to the caller of the main() method. Before this happens, the finally block is executed.

© RCET, Bhilai

Review Question

Q. 6 Which digits, and in what order, will be printed when the following program is run?

```
public class MyClass {
public static void main(String[] args) throws InterruptedException {
try {
f();
System.out.println("1");
} finally {
System.out.println("2");
}
System.out.println("3");
}
// InterruptedException is a direct subclass of Exception.
static void f() throws InterruptedException {
throw new InterruptedException("Time to go home.");
}
}
```

Select the one correct answer.

(a) The program will print 2 and throw InterruptedException.
(b) The program will print 1 and 2, in that order.
(c) The program will print 1, 2, and 3, in that order.
(d) The program will print 2 and 3, in that order.
(e) The program will print 3 and 2, in that order.
(f) The program will print 1 and 3, in that order..

Answer: (a)

© RCET, Bhilai

RUNGTA

User Define Exception Subclasses

- User can define own exception types to handle situations specific to their applications.
- It can be defined a subclass of **Exception (**which is a subclass of **Throwable)**
- all exceptions, including those that user create, have the methods defined by **Throwable** available to them**.**

© RCET, Bhilai

The Methods Defined by Throwable

Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned. Added by Java 2, version 1.4.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name. Added by Java 2, version 1.4
Throwable initCause(Throwable *causeExc*)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception. Added by Java 2, version 1.4

© RCET, Bhilai

RUNGTA

The Methods Defined by Throwable…

Method	Description
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.
void setStackTrace(StackTraceElement *elements*[ ])	Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use. Added by Java 2, version 1.4
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

© RCET, Bhilai

Example of user define Exception

```
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return "MyException[" + detail + "]";
} }
class ExceptionDemo {
static void compute(int a) throws MyException {
System.out.println("Called compute(" + a + ")");
if(a > 10)
throw new MyException(a);
System.out.println("Normal exit");
}
public static void main(String args[]) {
try {
compute(1);
compute(20);
} catch (MyException e) {
System.out.println("Caught " + e);
} } }
```


© RCET, Bhilai

RUNGTA

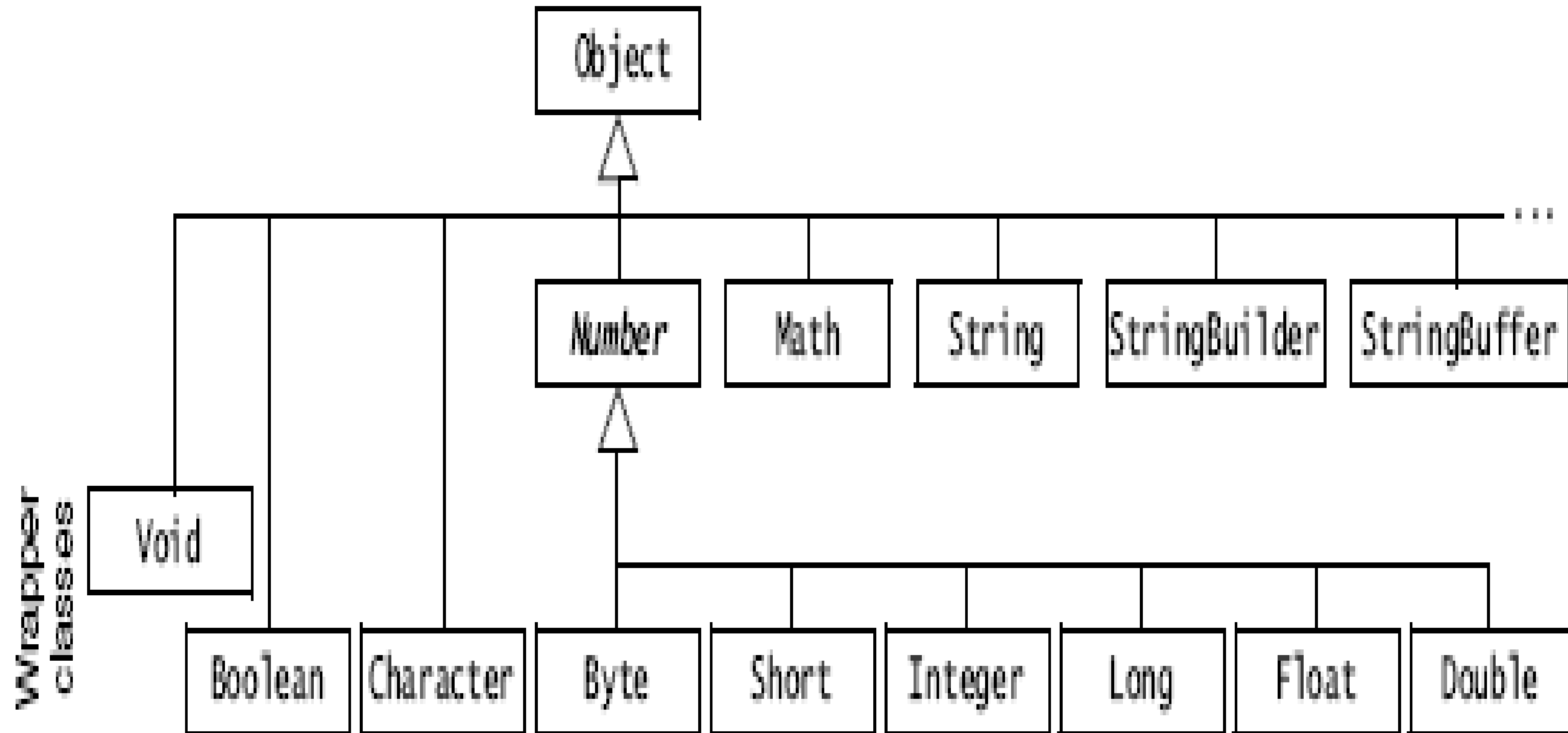
java.lang package

- The java.lang package is indispensable when programming in Java. It is automatically imported into every source file at compile time.
- The package contains the **Object** class that is the superclass of all classes.
- The wrapper classes (**Boolean, Character, Byte, Short, Integer, Long, Float, Double**) used to handle primitive values as objects.
- It provides classes essential for interacting with the JVM (Runtime), for security (**SecurityManager**), for loading classes (**ClassLoader**), for dealing with threads (**Thread**), and for exceptions (Throwable).
- The java.lang package also contains classes that provide the standard input, output, and error streams (**System**),
- String handling (**String, StringBuilder, StringBuffer**), and Mathematical functions (Math).

© RCET, Bhilai

RUNGTA

Partial Inheritance Hierarchy in the java.lang Package



© RCET, Bhilai

RUNGTA

java.lang includes the following classes:

- Boolean
- Long
- Byte
- Integer
- Short
- Void
- Math
- StrictMath (Java 2,1.3)
- Character
- Number
- String
- StringBuffer
- StringBuilder
- Class
- Object
- StackTraceElement (Java 2,1.4)
- ClassLoader
- Package (Java 2) System
- Compiler
- Process
- Thread
- Double
- Runtime
- ThreadGroup
- Float
- RuntimePermission (Java 2)
- ThreadLocal (Java 2)
- InheritableThreadLocal (Java 2) SecurityManager
- Throwable
- .....and more

© RCET, Bhilai

java.lang interfaces

- Cloneable
- Comparable
- Runnable
- CharSequence

© RCET, Bhilai

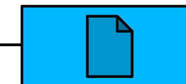
The Object Class

- All classes extend the Object class, either directly or indirectly.
- A class declaration, without the extends clause, implicitly extends the Object class.
- The Object class is always at the root of any inheritance hierarchy.
- The Object class defines the basic functionality that all objects exhibit and all classes inherit.

© RCET, Bhilai

The Methods Defined by Object

Method	Description
Object clone( ) throws CloneNotSupportedException	Creates a new object that is the same as the invoking object.
boolean equals(Object *object*)	Returns **true** if the invoking object is equivalent to *object*.
void finalize( ) throws Throwable	Default **finalize( )** method. This is usually overridden by subclasses.
final Class getClass( )	Obtains a **Class** object that describes the invoking object.
int hashCode( )	Returns the hash code associated with the invoking object.
final void notify( )	Resumes execution of a thread waiting on the invoking object.
final void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
final void wait( ) throws InterruptedException	Waits on another thread of execution.
final void wait(long *milliseconds*) throws InterruptedException	Waits up to the specified number of *milliseconds* on another thread of execution.
final void wait(long *milliseconds*, int *nanoseconds*) throws InterruptedException	Waits up to the specified number of *milliseconds* plus *nanoseconds* on another thread of execution.



© RCET, Bhilai

RUNGTA

Review Question

Q.7 What is the return type of the hashCode() method in the Object class?

Select the one correct answer.

(a) String

(b) int

(c) long

(d) Object

(e) Class

Answer : (b)

© RCET, Bhilai

RUNGTA

Review Question

Q.8 Which statement is true?

Select the one correct answer.

(a) If the references x and y denote two different objects, the expression x.equals(y) is always false.

(b) If the references x and y denote two different objects, the expression (x.hashCode() == y.hashCode()) is always false.

(c) The hashCode() method in the Object class is declared final.

(d) The equals() method in the Object class is declared final.

(e) All arrays have a method named clone.

Answer : (e)

© RCET, Bhilai

RUNGTA

Review Question

Q.9 Which exception can the clone() method of the Object class throw?

Select the one correct answer.

(a) CloneNotSupportedException

(b) NotCloneableException

(c) IllegalCloneException

(d) NoClonesAllowedException

Answer : (a)

© RCET, Bhilai

The Wrapper Classes

- Primitive values in Java are not objects. In order to manipulate these values as objects, the java.lang package provides a ***wrapper*** class for each of the primitive data types.
- All wrapper classes are final.
- The objects of all wrapper classes that can be instantiated are ***immutable****, i.e., the value in the* wrapper object cannot be changed.
- The **Void** class is considered a wrapper class, it does not wrap any primitive value and is not instantiable. It just denotes the Class object representing the keyword void.

© RCET, Bhilai

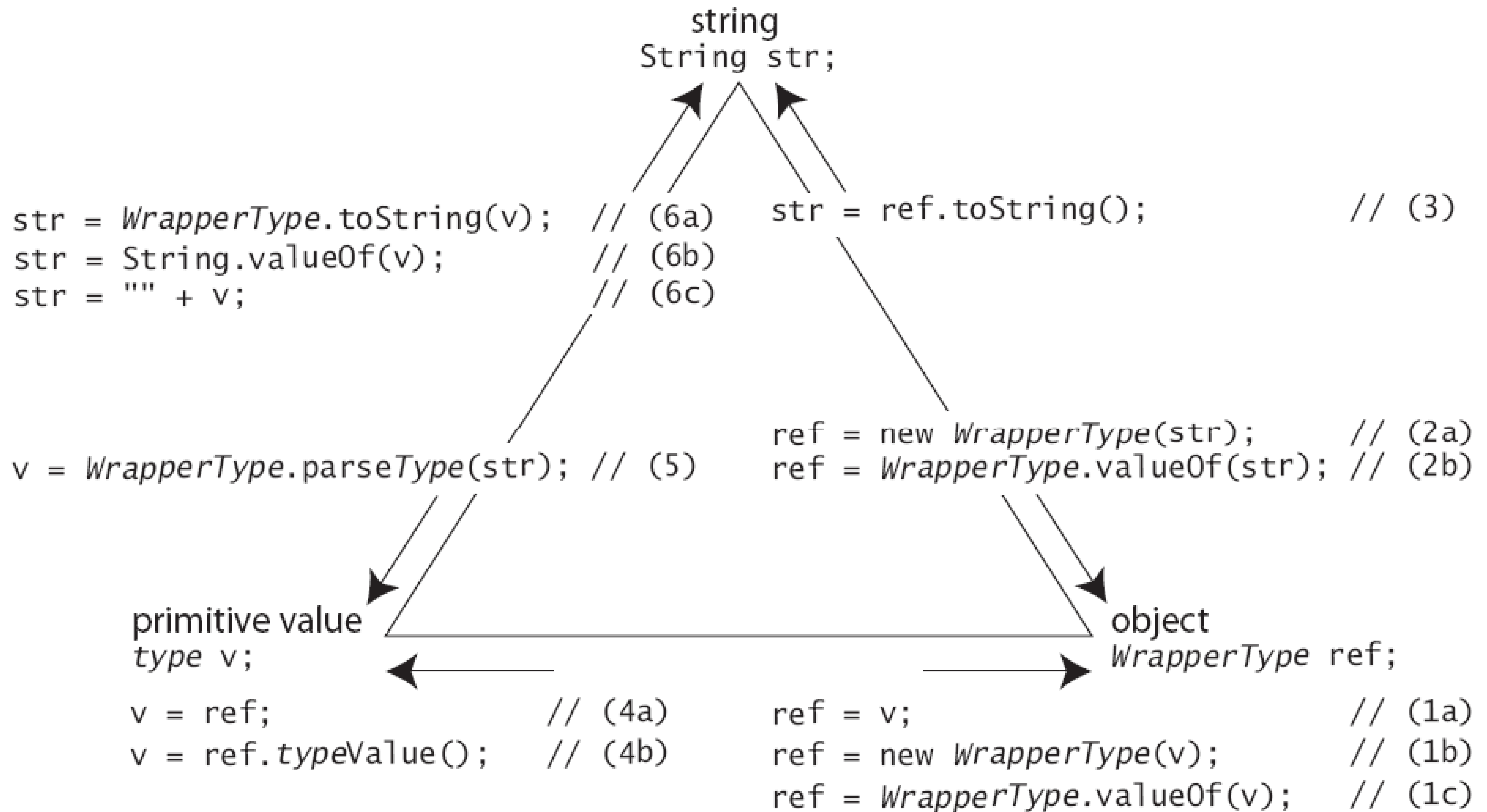
RUNGTA

The Wrapper Classes..

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

© RCET, Bhilai

Converting Values Between Primitive, Wrapper, and String Types



© RCET, Bhilai

RUNGTA

Common Wrapper Class Constructors

- The Character class has only one public constructor, taking a char value as parameter.
- The other wrapper classes all have two public one-argument constructors: one takes a primitive value and the other takes a string.
  - *WrapperType(type v)*
  - *WrapperType(String str)*
- Example
  - **Integer** intObj = new **Integer**(100);
  - **Integer** intObj1 = new **Integer**("125");
  - **Float** floatObj = new **Float**(255.4f);
  - **Double** doubleObj = new **Double**(2345.50);
  - **Character** charObj1 = '\n';
  - **Boolean** boolObj1 = true;
  - **Integer** intObj2 = 2008;
  - **Double** doubleObj1 = 3.14;

© RCET, Bhilai

RUNGTA

Float and Double Wrapper Class

- Double and Float are wrappers for floating-point values of type double and float, respectively.
- Both **Float and Double define the following constants:**

Constant	**Description**
MAX_VALUE	Maximum positive value
MIN_VALUE	Minimum positive value
NaN	Not a number
POSITIVE_INFINITY	Positive infinity
NEGATIVE_INFINITY	Negative infinity
TYPE	The **Class object for float or double**

© RCET, Bhilai

Methods of Float Class

Method	Description
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(float *num1*, float *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*. (Added by Java 2, version 1.4)
int compareTo(Float *f*)	Compares the numerical value of the invoking object with that of *f*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value. (Added by Java 2)
int compareTo(Object *obj*)	Operates identically to **compareTo(Float)** if *obj* is of class **Float**. Otherwise, throws a **ClassCastException**. (Added by Java 2)
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *FloatObj*)	Returns **true** if the invoking **Float** object is equivalent to *FloatObj*. Otherwise, it returns **false**.

© RCET, Bhilai

RUNGTA

Methods of Float Class...

Method	Description
static int floatToIntBits(float *num*)	Returns the IEEE-compatible, single-precision bit pattern that corresponds to the *num*.
float floatValue( )	Returns the value of the invoking object as a float.
int hashCode( )	Returns the hash code for the invoking object.
static float intBitsToFloat(int *num*)	Returns float equivalent of the IEEE-compatible, single-precision bit pattern specified by *num*.
int intValue( )	Returns the value of the invoking object as an int.
boolean isInfinite( )	Returns true if the invoking object contains an infinite value. Otherwise, it returns false.
static boolean isInfinite(float *num*)	Returns true if *num* specifies an infinite value. Otherwise, it returns false.
boolean isNaN( )	Returns true if the invoking object contains a value that is not a number. Otherwise, it returns false.
static boolean isNaN(float *num*)	Returns true if *num* specifies a value that is not a number. Otherwise, it returns false.
long longValue( )	Returns the value of the invoking object as a long.
static float parseFloat(String *str*) throws NumberFormatException	Returns the float equivalent of the number contained in the string specified by *str* using radix 10. (Added by Java 2)
short shortValue( )	Returns the value of the invoking object as a short.
String toString( )	Returns the string equivalent of the invoking object.
static String toString(float *num*)	Returns the string equivalent of the value specified by *num*.
static Float valueOf(String *str*) throws NumberFormatException	Returns the Float object that contains the value specified by the string in *str*.

© RCET, Bhilai

RUNGTA

Methods of Double Class...

Method	Description
byte byteValue( )	Returns the value of the invoking object as a byte.
static int compare(double *num1*, double *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*. (Added by Java 2, version 1.4)
int compareTo(Double *d*)	Compares the numerical value of the invoking object with that of *d*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value. (Added by Java 2)
int compareTo(Object *obj*)	Operates identically to compareTo(Double) if *obj* is of class Double. Otherwise, throws a ClassCastException. (Added by Java 2)
static long doubleToLongBits(double *num*)	Returns the IEEE-compatible, double-precision bit pattern that corresponds to the *num*.
double doubleValue( )	Returns the value of the invoking object as a double.
boolean equals(Object *DoubleObj*)	Returns true if the invoking Double object is equivalent to *DoubleObj*. Otherwise, it returns false.
float floatValue( )	Returns the value of the invoking object as a float.
int hashcode( )	Returns the hash code for the invoking object.

© RCET, Bhilai

Methods of Double Class...

Method	Description
int intValue( )	Returns the value of the invoking object as an int.
boolean isInfinite( )	Returns true if the invoking object contains an infinite value. Otherwise, it returns false.
static boolean isInfinite(double *num*)	Returns true if *num* specifies an infinite value. Otherwise, it returns false.
boolean isNaN( )	Returns true if the invoking object contains a value that is not a number. Otherwise, it returns false.
static boolean isNaN(double *num*)	Returns true if *num* specifies a value that is not a number. Otherwise, it returns false.
static double longBitsToDouble(long *num*)	Returns double equivalent of the IEEE-compatible, double-precision bit pattern specified by *num*.
long longValue( )	Returns the value of the invoking object as a long.
static double parseDouble(String *str*) throws NumberFormatException	Returns the double equivalent of the number contained in the string specified by *str* using radix 10. (Added by Java 2)
short shortValue( )	Returns the value of the invoking object as a short.
String toString( )	Returns the string equivalent of the invoking object.
static String toString(double *num*)	Returns the string equivalent of the value specified by *num*.
static Double valueOf(String *str*) throws NumberFormatException	Returns a Double object that contains the value specified by the string in *str*.

© RCET, Bhilai

RUNGTA

Demonstrate isInfinite() and isNaN()

```
class InfNaN {
    public static void main(String args[]) {
    Double d1 = new Double(1/0.);
    Double d2 = new Double(0/0.);
    System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1.isNaN());
    System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isNaN());
    System.out.println(Double.POSITIVE_INFINITY);
    System.out.println(Double.NEGATIVE_INFINITY);
} }
```


© RCET, Bhilai

RUNGTA

Byte, Short, Integer, and Long

- The following constants are defined:
  - MIN_VALUE Minimum value
  - MAX_VALUE Maximum value

© RCET, Bhilai

Methods in Integer Class

Method	Description
byte byteValue( )	Returns the value of the invoking object as a byte.
int compareTo(Integer *i*)	Compares the numerical value of the invoking object with that of *i*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value. (Added by Java 2)
int compareTo(Object *obj*)	Operates identically to **compareTo(Integer)** if *obj* is of class **Integer**. Otherwise, throws a ClassCastException. (Added by Java 2)
static Integer decode(String *str*) throws NumberFormatException	Returns an Integer object that contains the value specified by the string in *str*.
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *IntegerObj*)	Returns **true** if the invoking **Integer** object is equivalent to *IntegerObj*. Otherwise, it returns **false**.
float floatValue( )	Returns the value of the invoking object as a **float**.
static Integer getInteger(String *propertyName*)	Returns the value associated with the environmental property specified by *propertyName*. A null is returned on failure.
static Integer getInteger(String *propertyName*, int *default*)	Returns the value associated with the environmental property specified by *propertyName*. The value of *default* is returned on failure.

© RCET, Bhilai

Methods in Integer Class

Method	Description
static Integer getInteger(String *propertyName*, Integer *default*)	Returns the value associated with the environmental property specified by *propertyName*. The value of *default* is returned on failure.
int hashCode( )	Returns the hash code for the invoking object.
int intValue( )	Returns the value of the invoking object as an int.
long longValue( )	Returns the value of the invoking object as a long.
static int parseInt(String *str*) throws NumberFormatException	Returns the integer equivalent of the number contained in the string specified by *str* using radix 10.
static int parseInt(String *str*, int *radix*) throws NumberFormatException	Returns the integer equivalent of the number contained in the string specified by *str* using the specified *radix*.
short shortValue( )	Returns the value of the invoking object as a short.
static String toBinaryString(int *num*)	Returns a string that contains the binary equivalent of *num*.
static String toHexString(int *num*)	Returns a string that contains the hexadecimal equivalent of *num*.
static String toOctalString(int *num*)	Returns a string that contains the octal equivalent of *num*.
String toString( )	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(int *num*)	Returns a string that contains the decimal equivalent of *num*.

© RCET, Bhilai

RUNGTA

Methods in Integer Class

Method	Description
static String toString(int *num*, int *radix*)	Returns a string that contains the decimal equivalent of *num* using the specified *radix*.
static Integer valueOf(String *str*) throws NumberFormatException	Returns an **Integer** object that contains the value specified by the string in *str*.
static Integer valueOf(String *str*, int *radix*) throws NumberFormatException	Returns an **Integer** object that contains the value specified by the string in *str* using the specified *radix*.

© RCET, Bhilai

Example of Integer Class

```
public class IntegerRepresentation {
  public static void main(String[] args) {
    int positiveInt = +41;    // 051, 0x29
    int negativeInt = -41;    // 037777777727, -051, 0xffffffd7, -0x29

    System.out.println("String representation for decimal value: " + positiveInt);
    integerStringRepresentation(positiveInt);
    System.out.println("String representation for decimal value: " + negativeInt);
    integerStringRepresentation(negativeInt);
  }

  public static void integerStringRepresentation(int i) {
    System.out.println("    Binary:\t" + Integer.toBinaryString(i));
    System.out.println("    Octal:\t"  + Integer.toOctalString(i));
    System.out.println("    Hex:\t"    + Integer.toHexString(i));
    System.out.println("    Decimal:\t"  + Integer.toString(i));

    System.out.println("    Using toString(int i, int base) method:");
    System.out.println("    Base 2:\t" + Integer.toString(i, 2));
    System.out.println("    Base 8:\t" + Integer.toString(i, 8));
    System.out.println("    Base 16:\t"  + Integer.toString(i, 16));
    System.out.println("    Base 10:\t"  + Integer.toString(i, 10));
  }
}
```


© RCET, Bhilai

RUNGTA

Character

- The **Character class** defines several constants, including the following**:**

- MAX_RADIX	The largest radix
- MIN_RADIX	The smallest radix
- MAX_VALUE	The largest character value
- MIN_VALUE	The smallest character value
- TYPE The	**Class object for char**

© RCET, Bhilai

Methods in Character Class

Method	Description
static boolean isDefined(char *ch*)	Returns **true** if *ch* is defined by Unicode. Otherwise, it returns **false**.
static boolean isDigit(char *ch*)	Returns **true** if *ch* is a digit. Otherwise, it returns **false**.
static boolean isIdentifierIgnorable(char *ch*)	Returns **true** if *ch* should be ignored in an identifier. Otherwise, it returns **false**.
static boolean isISOControl(char *ch*)	Returns **true** if *ch* is an ISO control character. Otherwise, it returns **false**.
static boolean isJavaIdentifierPart(char *ch*)	Returns **true** if *ch* is allowed as part of a Java identifier (other than the first character). Otherwise, it returns **false**.
static boolean isJavaIdentifierStart(char *ch*)	Returns **true** if *ch* is allowed as the first character of a Java identifier. Otherwise, it returns **false**.
static boolean isLetter(char *ch*)	Returns **true** if *ch* is a letter. Otherwise, it returns **false**.
static boolean isLetterOrDigit(char *ch*)	Returns **true** if *ch* is a letter or a digit. Otherwise, it returns **false**.
static boolean isLowerCase(char *ch*)	Returns **true** if *ch* is a lowercase letter. Otherwise, it returns **false**.
static boolean isMirrored(char *ch*)	Returns **true** if *ch* is a mirrored Unicode character. A mirrored character is one that is reversed for text that is displayed right-to-left. (Added by Java 2, version 1.4)
static boolean isSpaceChar(char *ch*)	Returns **true** if *ch* is a Unicode space character. Otherwise, it returns **false**.

© RCET, Bhilai

RUNGTA

Methods in Character Class

Method	Description
static boolean isTitleCase(char *ch*)	Returns **true** if *ch* is a Unicode titlecase character. Otherwise, it returns **false**.
static boolean isUnicodeIdentifierPart(char *ch*)	Returns **true** if *ch* is allowed as part of a Unicode identifier (other than the first character). Otherwise, it returns **false**.
static boolean isUnicodeIdentifierStart(char *ch*)	Returns **true** if *ch* is allowed as the first character of a Unicode identifier. Otherwise, it returns **false**.
static boolean isUpperCase(char *ch*)	Returns **true** if *ch* is an uppercase letter. Otherwise, it returns **false**.
static boolean isWhitespace(char *ch*)	Returns **true** if *ch* is whitespace. Otherwise, it returns **false**.
static char toLowerCase(char *ch*)	Returns lowercase equivalent of *ch*.
static char toTitleCase(char *ch*)	Returns titlecase equivalent of *ch*.
static char toUpperCase(char *ch*)	Returns uppercase equivalent of *ch*.

© RCET, Bhilai

RUNGTA

Math Class

- The **Math** class contains all the floating-point functions that are used for geometry and trigonometry, as well as several general-purpose methods.

- Math defines two double constants:

  - E (approximately 2.72)

  - PI (approximately 3.14).

© RCET, Bhilai

RUNGTA

Methods in Math Class

Method	Description
static double sin(double *arg*)	Returns the sine of the angle specified by *arg* in radians.
static double cos(double *arg*)	Returns the cosine of the angle specified by *arg* in radians.
static double tan(double *arg*)	Returns the tangent of the angle specified by *arg* in radians.
static double asin(double *arg*)	Returns the angle whose sine is specified by *arg*.
static double acos(double *arg*)	Returns the angle whose cosine is specified by *arg*.
static double atan(double *arg*)	Returns the angle whose tangent is specified by *arg*.
static double atan2(double *x*, double *y*)	Returns the angle whose tangent is *x*/*y*.

© RCET, Bhilai

RUNGTA

Methods in Math Class

Method	Description
static double exp(double *arg*)	Returns e to the *arg*.
static double log(double *arg*)	Returns the natural logarithm of *arg*.
static double pow(double *y*, double *x*)	Returns *y* raised to the *x*; for example, pow(2.0, 3.0) returns 8.0.
static double sqrt(double *arg*)	Returns the square root of *arg*.

© RCET, Bhilai

Methods in Math Class

Method	Description
static int abs(int *arg*)	Returns the absolute value of *arg*.
static long abs(long *arg*)	Returns the absolute value of *arg*.
static float abs(float *arg*)	Returns the absolute value of *arg*.
static double abs(double *arg*)	Returns the absolute value of *arg*.
static double ceil(double *arg*)	Returns the smallest whole number greater than or equal to *arg*.
static double floor(double *arg*)	Returns the largest whole number less than or equal to *arg*.
static int max(int *x*, int *y*)	Returns the maximum of *x* and *y*.
static long max(long *x*, long *y*)	Returns the maximum of *x* and *y*.
static float max(float *x*, float *y*)	Returns the maximum of *x* and *y*.
static double max(double *x*, double *y*)	Returns the maximum of *x* and *y*.
static int min(int *x*, int *y*)	Returns the minimum of *x* and *y*
static long min(long *x*, long *y*)	Returns the minimum of *x* and *y*.
static float min(float *x*, float *y*)	Returns the minimum of *x* and *y*.
static double min(double *x*, double *y*)	Returns the minimum of *x* and *y*.
static double rint(double *arg*)	Returns the integer nearest in value to *arg*.
static int round(float *arg*)	Returns *arg* rounded up to the nearest **int**.
static long round(double *arg*)	Returns *arg* rounded up to the nearest **long**.

© RCET, Bhilai

RUNGTA

Miscellaneous Math Methods

- static double IEEEremainder(double *dividend, double divisor)*
- static double random( )
- static double toRadians(double *angle)*
- static double toDegrees(double angle)

© RCET, Bhilai

RUNGTA

Demonstrate toDegrees() and toRadians().

```
class Angles {
public static void main(String args[]) {
        double theta = 120.0;
        System.out.println(theta + " degrees is " +
        Math.toRadians(theta) + " radians.");
        theta = 1.312;
        System.out.println(theta + " radians is " +
        Math.toDegrees(theta) + " degrees.");
}
}
```


© RCET, Bhilai

RUNGTA

Demonstrate Random()

```
public class TestRandom{

public static void main(String args[]){

      for(int i = 1; i<= 100; i++){

         System.out.println(Math.round(Math.random()*100));

}}}
```


© RCET, Bhilai

RUNGTA

String Handling

- Handling character sequences or sting is supported through three final classes:
  - String,
  - StringBuilder
  - StringBuffer
- **Immutability**
  - The **String class** implements immutable character strings, which are read-only once the string has been created and initialized.
  - It is not changeable.
- **Mutability**
- StringBuilder and StringBuffer classes implement dynamic character strings.
- The StringBuffer class is a *thread-safe version* of the StringBuilder class.

© RCET, Bhilai

RUNGTA

String Constructors

- The String class supports several constructors. To create an empty String, you call the default constructor.
  - String()
  - String(String str)
  - String(char[]ch)
  - String(char[]ch, int *startIndex, int numChars)*
  - String(byte *asciiChars[ ])*
  - String(byte *asciiChars[ ], int startIndex, int numChars)*
  - String(StringBuilder builder)
  - String(StringBuffer buffer)

© RCET, Bhilai

RUNGTA

String Length

- The length of a string is the number of characters that it contains.

  To obtain this value , call the length( ) method.

- E.g.,

```
char chars[] = { 'a', 'b', 'c' };

String s = new String(chars);

 System.out.println(s.length());
```


© RCET, Bhilai

RUNGTA

String Concatenation

- In java we can use "**+**" operator for concatenation, which concatenates two strings & produces a string object as a result.

E.g.,

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

- This displays the string : **He is 9 years old.**

- One practical use of string concatenation is found when you are creating very long strings. We can break them into smaller pieces, using the + to concatenate them.

© RCET, Bhilai

String Concatenation…

E.g.,

```
int age = 9;

String s = "He is " + age + " years old.";

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

- Here age is an int rather than another String, but the output produced is the same as before. This is because the int value in age is automatically converted into its string representation within a String object.

© RCET, Bhilai

RUNGTA

String Concatenation…

```
String S="four:" +2+2;

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

- Output:

  four:22

- Operator precedence causes the concatenation of "four" with the string equivalent of 2 to take place first. This result is then concatenated with the string equivalent of 2 a second time.

© RCET, Bhilai

String Concatenation…

```
String S= "four:"+(2+2);
System.out.println(S);
```

Output:

four: 4

© RCET, Bhilai

RUNGTA

String Methods

- **<u>charAt( )</u>**: By using this method we can able to extract a single character from a String.

  General form:

  char **charAt**(int *index)*

- The value of *index* must be nonnegative and specify a location within the string.
- charAt( ) returns the character at the specified location.
- E.g.,

  char ch;

  ch = "abc".charAt(1);

  assigns the value "**b**" to ch.

© RCET, Bhilai

RUNGTA

String Methods

- <u>**getChars()**</u>**:**

By using this method we can extract more than one character at a time.

General Form:

void getChars(int *sourceStart,* int *sourceEnd,* char *target[ ],* int *targetStart)*

- ***sourceStart*** specifies the index of the beginning of the substring .
- ***sourceEnd*** specifies an index that is one past the end of the desired substring.

  The substring contains the characters from *sourceStart through sourceEnd.*
- The array that will receive the characters is specified by *target*.
- The index within target at which the substring will be copied is passed in *targetStart*.

© RCET, Bhilai

String Methods

- **getBytes()**:
- By using this method we can able to convert a string in to byte array. So that we can able to write this array on to the output stream.
- It is most useful when we are exporting a String value in to an environment that does not support 16-bit Unicode characters. Because most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

General Form:

```
byte[ ] getBytes( )
```

E.g.,

```
String s="Hello How are Your";
byte b[]=s.getBytes();
```

- getBytes() is an alternative to getChars() which stores the characters in an array of bytes. It uses the default character-to-byte conversions provided by platform.

© RCET, Bhilai

RUNGTA

String Methods

- **toCharArray()**:
- By using this method we can able to convert the string in to new character array.

General form:

```
char[ ] toCharArray( )
```

E.g.,

```
String str="Rungta";
char ch[]=str.toCharArray();
for(int i=0;i<ch.length;i++)
{
  System.out.println(ch[i]);
}
```


© RCET, Bhilai

String Comparison methods

- **equals()**: Compares two strings for equality.

  General Form:

  boolean equals(Object *str*)

- *str* is the String object being compared with the invoking String object.
- It returns true if the strings contain the same characters in the same order, and false otherwise. The comparison is case-sensitive.

© RCET, Bhilai

RUNGTA

String Comparison methods

- **equalsIgnoreCase( )**:
- To perform a comparison that ignores case differences call equalsIgnoreCase().

When it compares two strings it considers A-Z to be the same as a-z.

General Form:

```
boolean equalsIgnoreCase(String str)
```

- *str* is the String object being compared with the invoking String object. *It* returns true if the strings contain the same characters in the same order, and false otherwise.

© RCET, Bhilai

RUNGTA

String Comparison methods

- **regionMatches( )**:
- This method compares a specific region inside a string with another specific region in another string.

General Form:

boolean regionMatches(int s*tartIndex, String str2,* int *str2StartIndex, int numChars)*

boolean regionMatches(boolean *ignoreCase,* int *startIndex, String str2,* int *str2StartIndex, int numChars)*

© RCET, Bhilai

RUNGTA

String Comparison methods

- *startIndex* specifies the index at which the region begins within the invoking String object.
- The String being compared is specified by *str2.*
- *The index* at which the comparison will start within *str2 is specified by str2StartIndex.*
- *The length* of the substring being compared is passed in *numChars.*
- In the second form, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant.

© RCET, Bhilai

RUNGTA

String Comparison methods

- **startsWith()**:
- This method determines whether a given String begins with a specified string.

General Form:

boolean startsWith(String *str)*

boolean startsWith(String s*tr, int startIndex)*

- *str* is the String being tested. If the string matches, true is returned Otherwise false is returned
- E.g.,

```
String str="Rungta Engineering College of Technology";
boolean b=str.startsWith("Rungta");
System.out.println(b);
```


© RCET, Bhilai

RUNGTA

String Comparison methods

- **endsWith()**:

  This method determines whether a given String ends with a specified string.

General Form:

boolean endsWith(String *str)*

© RCET, Bhilai

String Comparison methods

- E.g.,

```
String str="Rungta Engg. College";
boolean b=str.endsWith("College");
System.out.println(b);
```


© RCET, Bhilai

Equals() Vs ==

- The equals( ) method compares the characters inside a String object.

- The == operator compares two object references to see whether they refer to the same instance.

© RCET, Bhilai

String Comparison methods

- **compareTo( )**:

By using this method we can able to compare two String objects character by character. This type of comparison is called as Dictionary type checking.

General Form:

int compareTo(String *str)*

str is the String being compared with the invoking String.

© RCET, Bhilai

RUNGTA

String Comparison methods

Value	Meaning
Less than zero	The invoking string is less than *str*
Greater than zero	The invoking string is greater than *str*.
Zero	The two strings are equal.

© RCET, Bhilai

RUNGTA

String Comparison methods

- **compareToIgnore()**:

  General Form:

  int compareToIgnoreCase(String *str)*

This method returns the same results as compareTo( ) except that case differences are ignored.

© RCET, Bhilai

Searching Strings

- **indexOf( ):**

  Searches for the first occurrence of a character or substring. This method returns the index at which the character or substring was found or –1 on failure.

- To search for the first occurrence of a character

  int indexOf(int *ch)*

- To search for the first occurrence of a substring

  int indexOf(String *str)*

© RCET, Bhilai

RUNGTA

Searching Strings

- we can specify a starting point for the searching using these forms:

  int indexOf(int *ch, int startIndex)*

  int indexOf(String *str, int startIndex)*

- Here, *startIndex* specifies the index at which point the search begins*.*

© RCET, Bhilai

Searching Strings

- **lastIndexOf( )**:

  Searches for the last occurrence of a character or substring.

- To search for the last occurrence of a character

  int lastIndexOf(int *ch)*

- To search for the first or last occurrence of a substring, use

  int lastIndexOf(String *str)*

  str specifies the substring.

© RCET, Bhilai

RUNGTA

Searching Strings

int lastIndexOf(int *ch, int startIndex)*

int lastIndexOf(String *str, int startIndex)*

© RCET, Bhilai

RUNGTA

Modifying a String

- **substring( )**:

  We can extract a substring by using this method.

  String substring(int *startIndex)*

- Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.

© RCET, Bhilai

Modifying a String

String substring(int *startIndex, int endIndex)*

- Here, *startIndex* specifies the beginning index, and endIndex specifies the stopping point. The string returned contains all the characters from the beginning index to the ending index.

© RCET, Bhilai

Modifying a String

- E.g.,

```
String s1="Rungta Engg. College";
String str =s.subString(6);
String str1=s.subString(6,8);
System.out.println(str);
System.out.println(str1);
```


© RCET, Bhilai

RUNGTA

Modifying a String

- **<u>concat( )</u>**:

  By using this method we can concatenate two strings .

  String concat(String *str)*

- This method creates a new object that contains the invoking string with the contents of *str* appended to the end. *concat(*) performs the same function as +.

© RCET, Bhilai

RUNGTA

Modifying a String

- E.g.1:

```
String s1="one";
String s2=s1.concat("two");
```

- E.g.2:

```
String s1="one";
String s2="two";
System.out.println(s1.concat(s2));
```

In this case new String object will be created and it is handled by JVM.

© RCET, Bhilai

RUNGTA

Modifying a String

- **replace( )**:

  This method replaces all occurrences of one character in the invoking string with another character.

  String replace(char *original,* char *replacement)*

- Here, *original* specifies the character to be replaced by the character specified by *replacement.* The resulting string is returned.

© RCET, Bhilai

RUNGTA

Modifying a String

- E.g.,

```
String s = "Hello".replace('l', 'w');
```

puts the string “Hewwo” into **s.**

© RCET, Bhilai

RUNGTA

Modifying a String

- **replaceAll()**:

    By using this method a group of characters can be replaced.

replaceAll(String *original*, String *replacement*)

E.g.,

String s="Hello".replaceAll("ll","LLLL");

© RCET, Bhilai

RUNGTA

Modifying a String

- **replaceFirst()**:

  This method replaces first character.

replaceFirst(String *original*, String *replacement*)

E.g.,

String s="Hello How";

System.out.println(s.replaceFirst("he","how"));

© RCET, Bhilai

RUNGTA

Modifying a String

- **trim( )**:

  The trim( ) method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

```
String trim( )
```

E.g.,

```
String s="   Hello World   ".trim();
System.out.println(s);
```


© RCET, Bhilai

RUNGTA

Data Conversion Using valueOf( )

- **valueOf()**:

By using this method any data type will able to convert in to reasonable String representation. It is a static method that is overloaded within String for all of Java's built-in types, so that each type can be converted properly into a string.

static String valueOf(double *num)*

static String valueOf(long *num)*

static String valueOf(Object *ob)*

static String valueOf(char *chars[ ])*

© RCET, Bhilai

RUNGTA

Data Conversion Using valueOf( )

static String valueOf(char *chars[ ], int startIndex, int numChars)*

- *chars* is the array that holds the characters,
- *startIndex* is the index into the array of characters at which the desired substring begins and
- *numChars* specifies the length of the substring.

© RCET, Bhilai

- **toString()**:

  By using this method we can able to convert any object in to String object type.

String toString()

© RCET, Bhilai

RUNGTA

Changing the case of characters

- **toLowerCase( ):** converts all the characters in a string from uppercase to lowercase.

  String toLowerCase( )

- **toUpperCase( ):** converts all the characters in a string from lowercase to uppercase

  String toUpperCase( )

© RCET, Bhilai

RUNGTA

StringBuffer

- StringBuffer is a peer class of String that provides much of the functionality of strings.
- String represents fixed-length, immutable character sequences where as StringBuffer represents growable and writeable character sequences.

© RCET, Bhilai

RUNGTA

StringBuilder and StringBuffer

- StringBuffer may have characters and substrings inserted in the middle or appended to the end.
- StringBuffer will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

© RCET, Bhilai

RUNGTA

StringBuffer Constructors

- Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

StringBuffer()

Constructs a string buffer with no characters in it and an initial capacity specified by the length argument.

StringBuffer(int *length*)

© RCET, Bhilai

RUNGTA

StringBuffer Constructors

- Constructs a string buffer so that it represents the same sequence of characters as the string argument.

StringBuffer(String *str*)

© RCET, Bhilai

length() and capacity()

- **length()**:

  This method returns current length of this StringBuffer.

  int length();

- **capacity()**:

  This method returns total allocated space of this StringBuffer.

  int capacity();

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- **<u>ensureCapacity()</u>**:

  Ensures that the capacity of the buffer is at least equal to the specified minimum.

void ensureCapacity(int *minimumCapacity*)

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- **setLength()**:

  To set the length of the buffer with in a StringBuffer object use setLength() method.

  void setLength(int *len*)

  len specifies the length of the buffer.

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- When we increase the size of the buffer null characters are added to the end of the existing buffer.

- If we call setLength() method with a value less than the current value returned by length() method, then the characters stored beyond the new length will be lost.

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- **charAt():**

  The value of a single character can be obtained from a StringBuffer by using charAt() method.

  char charAt(int *index*)

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- **<u>setChatAt()</u>:** We can set the value of character with in a stringBuffer by using setChatAt() method.

  void setChatAt(int *index*, char ch)

  index must not specify a location beyond the end of the buffer.

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- **getChars()**:

  By using this method we can copy a substring of a StringBuffer into an array.

void getChars(int *sourceStart,* int *sourceEnd,*

char *target[ ],* int *targetStart)*

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- **append():**

  This method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object.

  StringBuffer append(String *str)*

  StringBuffer append(int *num)*

  StringBuffer append(Object *obj)*

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- E.g.,

```
String s;
int a=42;
StringBuffer sb= new StringBuffer(40);
s=sb.append(“a=“).append(a).append(“!”).toString();
System.out.println(s);
```

Output: a=42!

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- **insert()**:

  This method inserts one String in to another.

  StringBuffer insert(int *index,* String *str)*

  StringBuffer insert(int *index,* char *ch)*

  StringBuffer insert(int *index,* Object *obj)*

index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

© RCET, Bhilai

StringBuffer Methods

- E.g.,

```
StringBuffer sb= new StringBuffer("I Java!");
sb.insert(2,"Like");
System.out.println(sb);
```

Output: I Like Java!

© RCET, Bhilai

StringBuffer Methods

- **reverse()**:

  By using this method we can reverse the characters with in a String.

  StringBuffer reverse()

This method returns the reversed object on which it was called.

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- **delete()**:

  This method deletes a sequence of characters from the invoking object.

StringBuffer delete(int *startIndex,* int *endIndex)*

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- **deleteCharAt()**:

  StringBuffer deleteCharAt(int *loc)*

  This method deletes the character at the index specified by *loc* and returns resulting StringBuffer.

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- **replace()**:

  This method replaces one set of characters with another set inside a StringBuffer object.

StringBuffer replace(int *startIndex,* int *endIndex,*

String *str)*

© RCET, Bhilai

RUNGTA

StringBuffer Methods

- **substring( ) :**

  String substring(int *startIndex)*

  This returns a substring that starts at *startIndex* and runs to the end of the invoking StringBuffer object.

© RCET, Bhilai

RUNGTA

StringBuffer Methods

String substring(int *startIndex, int endIndex)*

This returns a substring that starts at *startIndex* and runs through *endIndex*.

© RCET, Bhilai

RUNGTA

Collection Framework

- A data structure is nothing but arranging the data in a particular order, In order to perform some operations on it.

- The Built-in data Structure are for storing the homogeneous elements only. But java language needs Heterogeneous element collection. So to satisfy our new requirements we can able to create our own data structure.

© RCET, Bhilai

RUNGTA

Collection Framework

- Collection is a group of objects.

- Collection Framework means unified architecture for representing and manipulating the collection classes.

- From jdk1.2 onwards a new concept is introduced in util package called as “Collection Framework”.

© RCET, Bhilai

RUNGTA

Collection Framework

- Collections are of 2 types

    1) Single value collections

    2) Double value collections

- Single value collections are used for storing the data (or) record where as double value collections are meant for comparison.

© RCET, Bhilai

RUNGTA

Collection Framework

Single value collection	**Double value collection**
Vector	HashTable
Stack	Properties
ArrayList	HashMap
LinkedList	LinkedHashMap
HashSet	TreeMap
LinkedHashSet	
TreeHashSet	

© RCET, Bhilai

RUNGTA

Vector

- The Vector class implements a growable array of objects.
- Vector class is by default supporting synchronized methods.
- Vector is a special class which always supporting jdk1.0 and jdk1.2 methods.

© RCET, Bhilai

RUNGTA

Vector Constructors

- Vector():

  By using this constructor a collection object will be created, by default which will occupy 10 elements space.

  E.g. Vector v= new Vector();

© RCET, Bhilai

Vector Constructors

- Vector(Collection c):

  Constructs a vector containing the elements of the specified collection.

E.g. Vector v1= new vector();

Vector v2= new Vector(v1);

© RCET, Bhilai

RUNGTA

Vector Constructors

- Vector(int intialCapacity):

  Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.

E.g. Vector v=new Vector(25);

© RCET, Bhilai

RUNGTA

Vector Constructors

- Vector(int initialCapacity, int capacityIncrement):

  By using this constructor we can able to specify the initial, incremental capacity.

Whenever the initial size is filled up, based on incremental capacity the collection size will be expanded.

E.g. Vector v=new vector(5,4);

© RCET, Bhilai

RUNGTA

Vector Methods

- **add():**

  Inserts the specified element at the specified position in this Vector.

  void add(int index, Object element)

  Appends the specified element to the end of this Vector.

  boolean add(Object o)

© RCET, Bhilai

RUNGTA

Vector Methods

- **addAll():**

Appends all of the elements in the specified Collection to the end of this Vector.

boolean addAll(Collection c)

Inserts all of the elements in in the specified Collection into this Vector at the specified position.

boolean addAll(int index, Collection c)

© RCET, Bhilai

RUNGTA

Vector Methods

- **addElement()**:

  Adds the specified component to the end of this vector, increasing its size by one.

  void addElement(Object obj)

© RCET, Bhilai

RUNGTA

Vector Methods

- **capacity()**:

  Returns the current capacity of this vector.

  int capacity()

E.g.
```
Vector v=new Vector(3);
System.out.println(v.capacity());
```


© RCET, Bhilai

RUNGTA

Vector Methods

- **clear()**:

  Removes all of the elements from this Vector.

  void clear()

- **clone()**:

  Returns a clone of this vector.

  Object clone()

© RCET, Bhilai

Vector Methods

- **<u>contains()</u>**:

  Tests if the specified object is a component in this vector.
  boolean contains(Object element)

- Returns true if and only if the specified object is the same as a component in this vector, as determined by the equals method; false otherwise.

© RCET, Bhilai

RUNGTA

Vector Methods

containsAll():

Returns true if this Vector contains all of the elements in the specified Collection.

boolean containsAll(Collection c).

© RCET, Bhilai

RUNGTA

Vector Methods

- **copyInto()**:

  Copies the components of this vector into the specified array.

  void copyInto(Object[] anArray)

- **elementAt()**:

  Returns the component at the specified index.

  Object elementAt(int index)

© RCET, Bhilai

RUNGTA

Vector Methods

- **<u>ensureCapacity()</u>**:

  Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.

  void ensureCapacity(int mincapacity)

- **<u>equals()</u>**:

  Compares the specified Object with this Vector for equality.

  boolean equals(Object o)

© RCET, Bhilai

RUNGTA

Vector Methods

- **firstElement()**:

  Returns the first component (the item at index 0) of this vector.

  Object firstElement()

- **get()**:

  Returns the element at the specified position in this Vector.

  Object get(int index)

© RCET, Bhilai

RUNGTA

Vector Methods

- **hashCode()**:

  Returns the hash code value for this Vector.

  int hashCode()

- **indexOf()**:

  Searches for the first occurrence of the given argument.

  int indexOf(Object element)

  Searches for the first occurrence of the given argument, beginning the search at index.

  int indexOf(Object element, int index)

© RCET, Bhilai

RUNGTA

Vector Methods

- **insertElementAt()**:

  Inserts the specified object as a component in this vector at the specified index.

  void insertElementAt(Object obj, int index)

- **lastElement()**:

  Returns the last component of the vector.

  Object lastElement()

© RCET, Bhilai

RUNGTA

Vector Methods

- **isEmpty()**:

  Tests if this vector has no components.

  boolean isEmpty()

- Returns true if and only if this vector has no components, that is, its size is zero; false otherwise.

© RCET, Bhilai

RUNGTA

Vector Methods

- **lastIndexOf()**:

Returns the index of the last occurrence of the specified object in this vector.

int lastIndexOf(Object element)

Searches backwards for the specified object, starting from the specified index, and returns an index to it.

int lastIndexOf(Object element,int index)

© RCET, Bhilai

RUNGTA

Vector Methods

- **remove()**:

Removes the element at the specified position in this Vector.

Object remove(int index)

Removes the first occurrence of the specified element in this Vector .

boolean remove(Object o)

Returns true if the Vector contained the specified element.

© RCET, Bhilai

RUNGTA

Vector methods

<u>removeAll()</u>:

Removes from this Vector all of its elements that are contained in the specified Collection.

boolean removeAll(Collection c)

<u>removeAllElements()</u>:

Removes all components from this vector and sets its size to zero.

void removeAllElements()

© RCET, Bhilai

RUNGTA

Vector Methods

- **removeElementAt()**:

  Deletes the component at the specified index.

  void removeElementAt(int index)

- **removeElement()**:

  Removes the first occurrence of the argument from this vector.

  boolean removeElement(Object obj)
  Returns true if the argument was a component of this vector; false otherwise.

© RCET, Bhilai

RUNGTA

Vector methods

- **removeRange()**:

  Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.

  protected void removeRange(int fromIndex,

  int toIndex)

- **retainAll()**:

  Retains only the elements in this Vector that are contained in the specified Collection.

  boolean retainAll(Collection c)

© RCET, Bhilai

RUNGTA

Vector methods

- **set()**:

  Replaces the element at the specified position in this Vector with the specified element.

  Object set(int index,Object element)

- **setElementAt()**:

  Sets the component at the specified index of this vector to be the specified object.

  void setElementAt(Object obj, int index)

© RCET, Bhilai

Vector Methods

- **setSize()**:

  Sets the size of this vector.

  void setSize(int newSize)

- **size()**:

  Returns the number of components in this vector.

  int size()

© RCET, Bhilai

RUNGTA

Vector Methods

- **toArray()**:

  Returns an array containing all of the elements in this Vector in the correct order.

  Object[] toArray()

© RCET, Bhilai

Vector Methods

- **toString()**:

  Returns a string representation of this Vector, containing the String representation of each element.

  String toString()

- **trimToSize()**:

  Trims the capacity of this vector to be the vector's current size.

  void trimToSize()

© RCET, Bhilai