Exception handling in Java

Yulia Newton, Ph.D.

CS151, Object Oriented Programming

San Jose State University

Spring 2020

Introduction to exceptions

- Exception is unwanted or unexpected run-time behavior
  - Normal flow of the program is disrupted
  - Without exception handling, the program that encounters an error terminates abnormally
  - Such terminations are undesirable, therefore we want to “handle” these events
- Exception handling is a mechanism for handling run-time errors
- A reasonably designed program should anticipate and catch exceptions

Reasons for exception events

- Exceptions might occur for any number of reasons
  - Invalid data was supplied
  - The specified file does not exist
  - Lost network connection
  - Invalid case of downcasting
  - Hardware failures
  - Insufficient resource
  - …
- Exceptions can be caused by users, physical resources, careless programming, etc.

Advantages of exception handling

- Exception handling allows maintaining a normal program flow
  - We want to avoid abnormal terminations as much as possible
- It might be problematic if some statements of the program execute but some do not
  - Could leave the system in unknown/unstable state
  - If exception handling is performed then all statements will be executed

How exceptions arise in Java

- If an error occurs
  - An exception object is created
  - Program execution halts
  - JRE attempts to locate appropriate handling of the particular exception type
- Exception object contains a lot of useful information about the event that created it
- We call it “throwing an exception”

Exception handling

- Exception handler is a block of code that handles an exception
- JRE applies the following logic for locating an appropriate exception handler
  - Start searching in the method where the error occurred
  - If no appropriate exception handler found then move to the method caller
  - Continue until exception handler is found (catching exception) or the program is terminated
- Java exception handling framework only handles run-time errors
  - Compile time errors are not handled by exception handling

Call stack

- Ordered list of methods that have been called to get to specific method
- For example, call stack looks like: **A() -> B() -> C()**

  Exception is raised in method C()

  JRE will search for exception handler in the following order: **C, B, A**

A few exception handling terms

- **try-catch** – exception handling is performed in a try-catch block
  - **try** – is the block where we find program instructions
    - Has to be followed by either **catch** or **finally**
  - **catch** – is the block where we handle the exceptions
- **throw** – sometimes it is advantageous to create an exception object in the code and "throw" it to the JRE to handle it
- **throws** – if we throw an exception and do not handle it in the method then we need to specify *throws* in the method declaration
- **finally** – optional block of code that is executed upon termination of the program, whether exception occurred or not
  - Can only be used with try-catch block
  - We might want to use it to close some resources

Difference between error and exception

- Program can catch exceptions but not errors
  - E.g. *NullPointerException*, *ArithmeticException*, *ArrayIndexOutOfBoundsException* exceptions can be caught with try-catch block
- Errors are used by JVM to indicate issues that have to do with JRE itself
  - E.g. *StackOverflowError*, *OutOfMemoryError*, *VirtualMachineError*, *AssertionError* errors cannot be caught
- Errors are considered more severe than exceptions and are not handled or caught
  - Irrecoverable

Checked vs. unchecked exception

- Classes that directly inherit from ***Throwable*** class, except *RuntimeException* and *Error* are checked exceptions
  - E.g. *IOException*, *SQLException*
  - Checked at compile-time
- Classes that inherit from *RuntimeException* class are unchecked exceptions
  - Checked at run-time
  - Are not checked at compile-time
  - You will never get a compile error forcing you to catch these

Try-catch blocks

```
try{
    // statement that might throw an exception
}catch(ExceptionType Object){
    // code that handles this exception
}
```

Example: catching divide by zero exception

```
public class Test
{
    public static void main(String args[])
    {
        try{
            int data=100/0;
        }catch(ArithmeticException e){
            System.out.println(e);
        }

        System.out.println("we can put code outside of try-catch block");
    }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
we can put code outside of try-catch block
```

Example: call stack is retraced to find matching exception handling

```
public class Test {
    static int performDivision(int x, int y){
        int result = x/y;
        return result;
    }

    static int divide(int x, int y){
        int result =0;

        try{
          result = performDivision(x,y);
        }catch(NumberFormatException ex){
           System.out.println("NumberFormatException occured");
           System.out.println(ex.getMessage());
        }
        return result;
    }

    public static void main(String args[]){
        int x = 1, y = 0;

        try{
            int i = divide(x,y);

        }catch(ArithmeticException ex) {
            System.out.println("ArithmeticException occured");
            System.out.println(ex.getMessage());
        }
    }
}
```

NumberFormatException does not match our exception

ArithmeticException matches our divide by zero exception

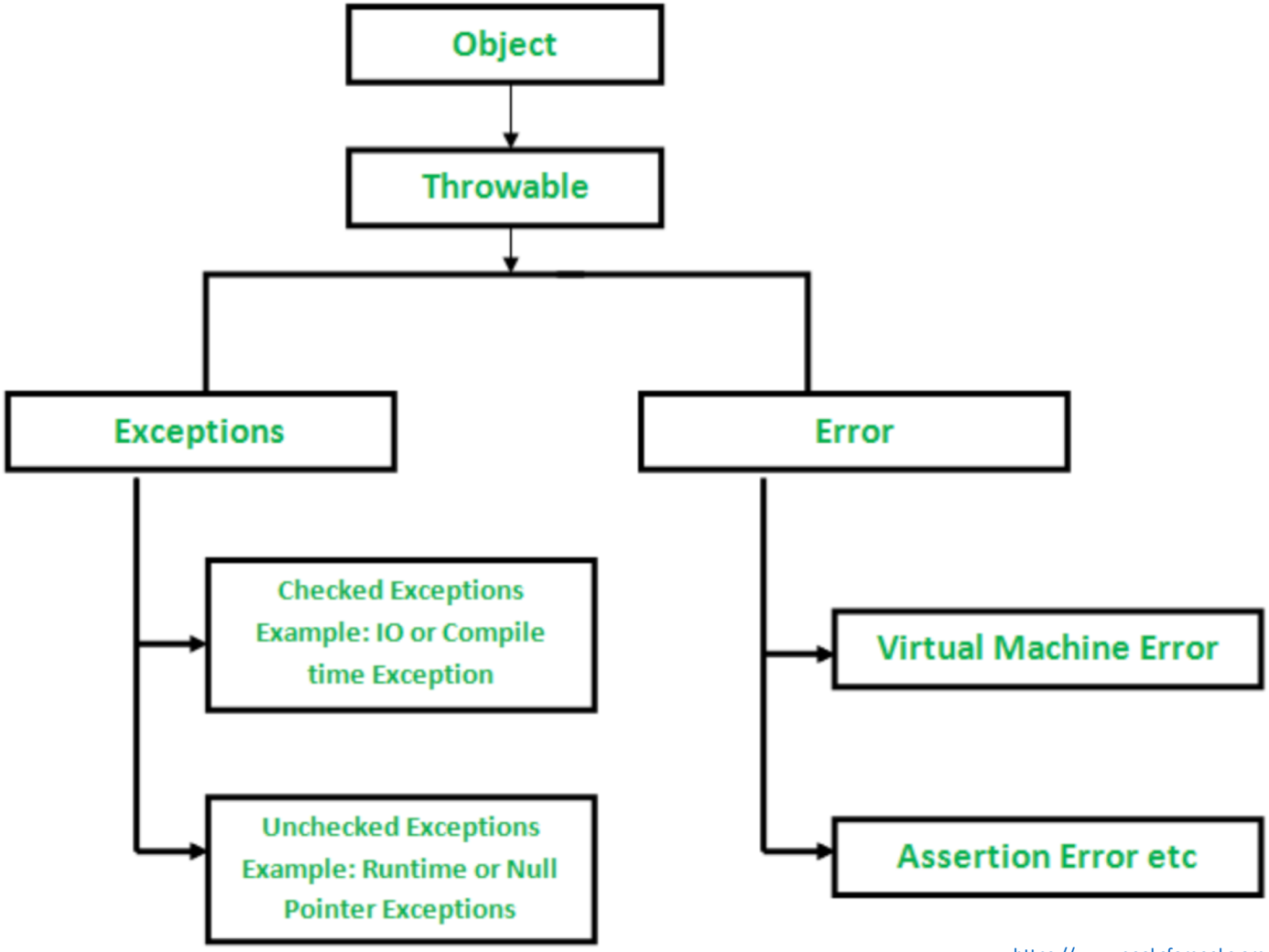
Output:

```
ArithmeticException occured
/ by zero
```

Call stack: main() -> divide() -> performDivision()

Exception type hierarchy

- Exception class identifies the kind of exception it is
- Every exception type extends *java.lang.Exception* class
  - Every exception type is a subclass and therefore ”is” an Exception
- There is a hierarchies and groups within exceptions
  - E.g. NumberFormatException is a subtype of IllegalArgumentException



https://www.geeksforgeeks.org

Default exception

- Even with the best of planning sometimes unforeseen exceptions might arise
- Sometimes it could be a good idea to catch a generic exception
  - *Exception* class or *RuntimeException* class
  - *Throwable* class
- Use judiciously
  - Best practice is to be as specific with your exceptions as possible

Example: catching a default exception

```java
public class Test
{
    public static void main(String args[])
    {
        try{
            int data=100/0;
        }catch(ArithmeticException e){
            System.out.println(e);
        }
        catch(Exception e){
            System.out.println("We caught an except of unanticipated type");
        }

        System.out.println("we can put code outside of try-catch block");
    }
}
```

```
java.lang.ArithmeticException: / by zero
we can put code outside of try-catch block
```

```java
public class Test
{
    public static void main(String args[])
    {
        try{
            String str = null;
            System.out.println(str.length());
        }catch(ArithmeticException e){
            System.out.println(e);
        }
        catch(Exception e){
            System.out.println("We caught an except of unanticipated type");
        }

        System.out.println("we can put code outside of try-catch block");
    }
}
```

```
We caught an except of unanticipated type
we can put code outside of try-catch block
```

What happens if we do not anticipate and catch an exception?

From previous example:

```
public class Test
{
    public static void main(String args[])
    {
        try{
            String str = null;
            System.out.println(str.length());
        }catch(ArithmeticException e){
            System.out.println(e);
        }

        System.out.println("we can put code outside of try-catch block");
    }
}
```

We do not have exception handling for the matching exception or the default exception handling

```
Exception in thread "main" java.lang.NullPointerException
        at Test.main(Test.java:1189)
```

Catching multiple exceptions in the same try-catch block

- Multiple exceptions can be caught with a single try block
- Just sequentially add individual catch block for each type of exception you anticipate to catch
- Don’t forget to add default exception
  - Not required, just a good practice

Example: catching multiple exceptions

```java
class ImportantClass {
    public void doOperation(String[] myList, String newVal, int denominator){
        try{
            int numerator;

            myList[5] = newVal;
            numerator = Integer.parseInt(myList[4]);
            System.out.println("Results of the division operation is: "+numerator/denominator);
        }catch(ArithmeticException e){
            System.out.println("ArithmeticException:");
            System.out.println(e);
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("ArrayIndexOutOfBoundsException:");
            System.out.println(e);
        }catch(NullPointerException e){
            System.out.println("NumberFormatException:");
            System.out.println(e);
        }catch(Exception e){
            System.out.println("Default exception:");
            System.out.println(e);
        }

        System.out.println("Operations outside of try-catch block");
    }
}
```

```java
public class Test {
    public static void main(String args[]){
        String[] myList = {"1","2","3","4","5","6"};
        ImportantClass c = new ImportantClass();

        c.doOperation(myList, "5", 1);
    }
}
```

```
Results of the division operation is: 5
Operations outside of try-catch block
```

Example: catching multiple exceptions

```
class ImportantClass {
    public void doOperation(String[] myList, String newVal, int denominator){
        try{
            int numerator;

            myList[5] = newVal;
            numerator = Integer.parseInt(myList[4]);
            System.out.println("Results of the division operation is: "+numerator/denominator);
        }catch(ArithmeticException e){
            System.out.println("ArithmeticException:");
            System.out.println(e);
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("ArrayIndexOutOfBoundsException:");
            System.out.println(e);
        }catch(NullPointerException e){
            System.out.println("NumberFormatException:");
            System.out.println(e);
        }catch(Exception e){
            System.out.println("Default exception:");
            System.out.println(e);
        }

        System.out.println("Operations outside of try-catch block");
    }
}
```

```
public class Test {
    public static void main(String args[]){
        String[] myList = {"1","2","3","4"};
        ImportantClass c = new ImportantClass();

        c.doOperation(myList, "5", 1);
    }
}
```

```
java.lang.ArrayIndexOutOfBoundsException: 5
Operations outside of try-catch block
```

Example: catching multiple exceptions

```
class ImportantClass {
    public void doOperation(String[] myList, String newVal, int denominator){
        try{
            int numerator;

            myList[5] = newVal;
            numerator = Integer.parseInt(myList[4]);
            System.out.println("Results of the division operation is: "+numerator/denominator);
        }catch(ArithmeticException e){
            System.out.println("ArithmeticException:");
            System.out.println(e);
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("ArrayIndexOutOfBoundsException:");
            System.out.println(e);
        }catch(NullPointerException e){
            System.out.println("NumberFormatException:");
            System.out.println(e);
        }catch(Exception e){
            System.out.println("Default exception:");
            System.out.println(e);
        }

        System.out.println("Operations outside of try-catch block");
    }
}
```

```
public class Test {
    public static void main(String args[]){
        String[] myList = {"1","2","3","4"};
        ImportantClass c = new ImportantClass();

        c.doOperation(myList, "5", 1);
    }
}
java.lang.ArrayIndexOutOfBoundsException: 5
Operations outside of try-catch block
```

```
public class Test {
    public static void main(String args[]){
        String[] myList = {"1","2","3","4","a","5"};
        ImportantClass c = new ImportantClass();

        c.doOperation(myList, "5", 1);
    }
}
java.lang.NumberFormatException: For input string: "a"
Operations outside of try-catch block
```

Example: catching multiple exceptions

```
class ImportantClass {
    public void doOperation(String[] myList, String newVal, int denominator){
        try{
            int numerator;

            myList[5] = newVal;
            numerator = Integer.parseInt(myList[4]);
            System.out.println("Results of the division operation is: "+numerator/denominator);
        }catch(ArithmeticException e){
            System.out.println("ArithmeticException:");
            System.out.println(e);
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("ArrayIndexOutOfBoundsException:");
            System.out.println(e);
        }catch(NullPointerException e){
            System.out.println("NumberFormatException:");
            System.out.println(e);
        }catch(Exception e){
            System.out.println("Default exception:");
            System.out.println(e);
        }

        System.out.println("Operations outside of try-catch block");
    }
}
```

```
public class Test {
    public static void main(String args[]){
        String[] myList = {"1","2","3","4"};
        ImportantClass c = new ImportantClass();

        c.doOperation(myList, "5", 1);
    }
}
```

```
java.lang.ArrayIndexOutOfBoundsException: 5
Operations outside of try-catch block
```

```
public class Test {
    public static void main(String args[]){
        String[] myList = {"1","2","3","4","a","5"};
        ImportantClass c = new ImportantClass();

        c.doOperation(myList, "5", 1);
    }
}
```

```
java.lang.NumberFormatException: For input string: "a"
Operations outside of try-catch block
```

```
public class Test {
    public static void main(String args[]){
        String[] myList = {"1","2","3","4",null,"5"};
        ImportantClass c = new ImportantClass();

        c.doOperation(myList, "5", 1);
    }
}
```

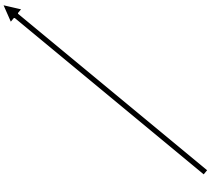
```
Default exception:
java.lang.NumberFormatException: null
Operations outside of try-catch block
```

Another way to catch multiple exceptions

```
class ImportantClass {
    public void doOperation(String[] myList, String newVal, int denominator){
        try{
            int numerator;

            myList[5] = newVal;
            numerator = Integer.parseInt(myList[4]);
            System.out.println("Results of the division operation is: "+numerator/denominator);
        }catch(ArithmeticException | ArrayIndexOutOfBoundsException | NumberFormatException e){
            System.out.println(e);
        }catch(Exception e){
            System.out.println("Default exception:");
            System.out.println(e);
        }

        System.out.println("Operations outside of try-catch block");
    }
}
```



Exception types are separated by |

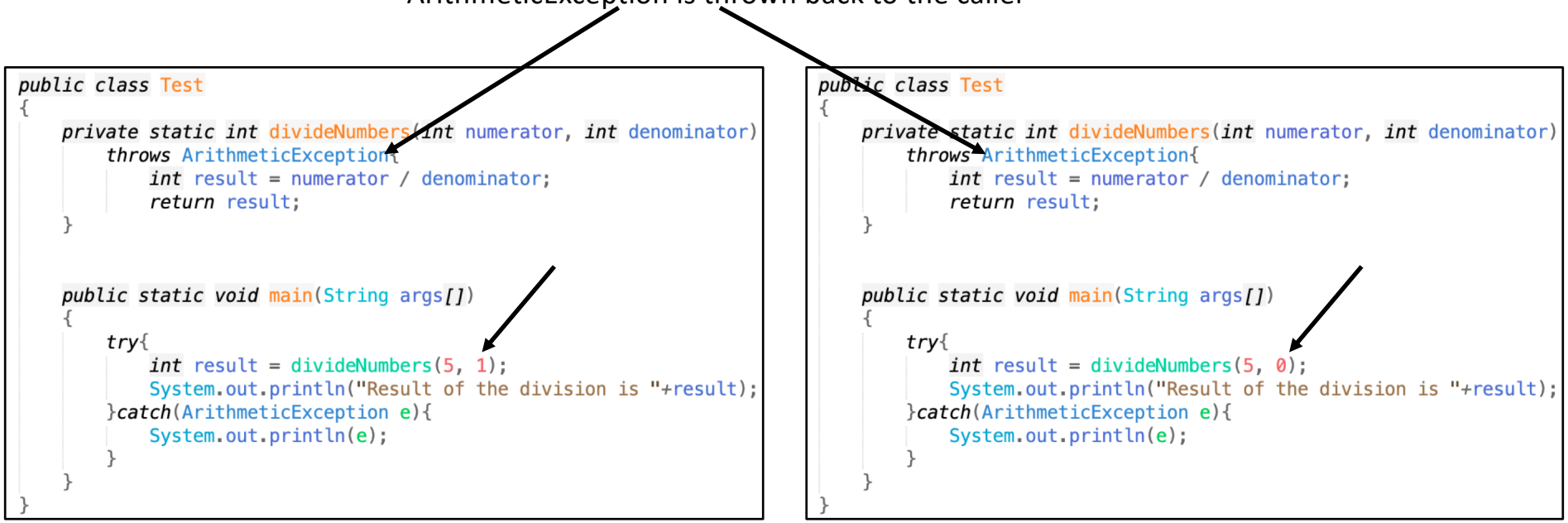
throws is a way to delegate exception handling to the caller

- Sometimes we want the caller to handle possible exceptions
- *throws* specifies the exceptions that the method might throw
  - The method does not catch those exceptions inside its code
- Can only propagate checked exceptions
- Can specify multiple exceptions

```
public void divideNumbers(int numerator, int denominator)
    throws ArithmeticException{
        // statements go here
    }
```

Example: throw exception back to the caller

ArithmeticException is thrown back to the caller



```java
public class Test
{
    private static int divideNumbers(int numerator, int denominator)
        throws ArithmeticException{
            int result = numerator / denominator;
            return result;
    }


    public static void main(String args[])
    {
        try{
            int result = divideNumbers(5, 1);
            System.out.println("Result of the division is "+result);
        }catch(ArithmeticException e){
            System.out.println(e);
        }
    }
}
```

```
Result of the division is 5
```

```java
public class Test
{
    private static int divideNumbers(int numerator, int denominator)
        throws ArithmeticException{
            int result = numerator / denominator;
            return result;
    }


    public static void main(String args[])
    {
        try{
            int result = divideNumbers(5, 0);
            System.out.println("Result of the division is "+result);
        }catch(ArithmeticException e){
            System.out.println(e);
        }
    }
}
```

```
java.lang.ArithmeticException: / by zero
```

throw raises an exception event

```
public void doSomething(){
    ...
    throw new ExceptionType();
    ...
}
```

- Transfers control of *try* block to *catch* block
- Sometimes logically it makes sense to throw an exception even if one did not arise
  - Throws exception explicitly
  - Throws one exception at a time
- Syntax includes instantiating a new instance of Exception or its subtype class
- Can only propagate unchecked exceptions
- Try to be as specific as possible with the type of exception you are throwing

Example: *throw* exception to catch block

```
abstract class Animal{
    abstract void sound();
}

class Cat extends Animal {
    public void sound(){
        System.out.println("Meow");
    }
}

class Dog extends Animal {
    public void sound(){
        System.out.println("Bark");
    }

    public static void introduce(Animal a){
        try{
            if(a instanceof Dog){
                System.out.println("I am a dog");
            }else{
                throw new IllegalArgumentException();
            }
        }catch(IllegalArgumentException e){
            System.out.println(e);
        }
    }
}
```

```
public class Test
{
    public static void main(String args[])
    {
        Dog d = new Dog();
        Cat c = new Cat();

        System.out.println("Calling introduce and passing in a Dog object:");
        Dog.introduce(d);

        System.out.println("\nCalling introduce and passing in a Cat object:");
        Dog.introduce(c);
    }
}
```

```
Calling introduce and passing in a Dog object:
I am a dog

Calling introduce and passing in a Cat object:
java.lang.IllegalArgumentException
```

Custom exceptions

- Programmers can define their own exception types by extending *java.lang.Exception* class
- A good rule of thumb is that if you can use any of Java's standard exceptions do so
  - Does your custom exception provide any information or functionality not available in one of the standard exceptions?
- Following naming convention is a good practice
  - Class names end with "Exception"
- Provide Javadoc comments to document your custom exception

Custom checked exceptions

- Extend *Exception* class
- Provide constructor that sets causing exception
  - Not required but considered a good practice
- In the custom exception's constructor invoke the constructor of the superclass
- Often a good idea to overload the constructor

Example: custom checked exception

```
class InvalidNumberLegsException extends Exception{
    InvalidNumberLegsException(String s){
        super(s);
    }
}

class Dog {
    private int numLegs;

    Dog(){
        this.numLegs = 0;
    }

    public void setLegs(int n){
        try{
            if(n <= 0 || n > 10){
                throw new InvalidNumberLegsException("Invalid number of legs were specified");
            }else{
                System.out.println("Setting number of legs to "+n);
                this.numLegs = n;
            }
        }catch(InvalidNumberLegsException e){
            System.out.println(e);
        }
    }
}
```

can have as many overloaded constructors as we want

invoking the constructor of superclass with the message string

```
public class Test
{
    public static void main(String args[])
    {
        Dog d = new Dog();
        d.setLegs(4);
        d.setLegs(0);
        d.setLegs(20);
    }
}
```

Output:

```
Setting number of legs to 4
InvalidNumberLegsException: Invalid number of legs were specified
InvalidNumberLegsException: Invalid number of legs were specified
```

Custom unchecked exceptions

- Extend *RuntimeException* class
- Same remarks about the constructor in checked exceptions go for unchecked exceptions as well

Finally block

```
try{
    // statement that might throw an exception
}catch(ExceptionType Object){
    // code that handles this exception
}finally{
    // statements to execute before exiting
}
```

- Must be associated with a *try* block
  - However, it is optional
- If no exception occurs then *finally* block executes after the *try* block has completed
- If exception occurs then *finally* block is executed after the *catch* block
- An exception within *finally* block behaves as any other exception
- *finally* block executes even if there is a return or other transfer statements within *try* block
- Usually used to close up and release resources (e.g. streams, files, etc.)

Example: finally block is executed after the catch block

```java
public class Test
{
    public static void main(String args[])
    {
        try{
            int x = 5;
            int y = 0;
            int result = x / y;
        }catch(ArithmeticException e){
            System.out.println(e);
        }finally{
            System.out.println("finally block is executed");
        }

        System.out.println("Code outside of try-catch-finally block");
    }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
finally block is executed
Code outside of try-catch-finally block
```

Example: finally block is executed without matching exception handling

```java
public class Test
{
    public static void main(String args[])
    {
        try{
            int x = 5;
            int y = 0;
            int result = x / y;
        }catch(NullPointerException e){
            System.out.println(e);
        }finally{
            System.out.println("finally block is executed");
        }

        System.out.println("Code outside of try-catch-finally block");
    }
}
```

This exception type is not a match for the exception thrown in the try block

Output:

```
finally block is executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.main(Test.java:1281)
```

Example: finally block is executed without an exception being thrown

```
public class Test
{
    public static void main(String args[])
    {
        try{
            int x = 5;
            int y = 1;
            int result = x / y;
        }catch(NullPointerException e){
            System.out.println(e);
        }finally{
            System.out.println("finally block is executed");
        }

        System.out.println("Code outside of try-catch-finally block");
    }
}
```

No exception is thrown in this try block

Output:

```
finally block is executed
Code outside of try-catch-finally block
```

close() statement

- Used to close all open streams
- It is considered a good practice to use *close()* inside *finally* block

Example: closing stream resources in *finally* block

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class Test {

   public static void main(String args[]) {
      FileReader fr = null;
      String inputFile = "myFile.txt";

      try {
         File file = new File(inputFile);
         fr = new FileReader(file);
         char [] a = new char[50];
         fr.read(a);
         for(char c : a)
         System.out.print(c);   // prints the characters one by one
      } catch (IOException e) {
         e.printStackTrace();
      }finally {
         try {
            fr.close();
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }
   }
}
```

We close input reader before exiting

Inspired by example from https://www.tutorialspoint.com/

There are circumstanced under which *finally* block is not executed or finished

- Exception occurs within the *finally* block
- Death of the execution thread
- *System.exit()* method has been used

Useful commonly used exception types

- *NumberFormatException* – String cannot be converted as a number
- *IllegalArgumentException* – provided method argument is invalid
- *ArithmeticException* – division by zero exception
- *NullPointerException* – performing an operation on a variable with a null value
- *ArrayIndexOutOfBoundsException* – attempting to access a position in an array that doesn't exist
- *IOException* – something with the file system, network, or database failed
- *ClassCastException* – illegal casting was attempted

Useful common error types

- *StackOverflowError* – stack trace is too big; happens with large applications or problematic execution (e.g. infinite recursion)
- *NoClassDefFoundError* – class failed to load (e.g. invalid classpath)
- *OutOfMemoryError* – JVM does not have enough available memory; could happen due to memory leaks

Throwable

- *Throwable* is a superclass of all exception types
- It is not considered a good practice to catch/throw *Throwable* type

Useful public methods of Throwable class

- *public String getMessage()* – returns exception message string
- *public String getLocalizedMessage()* – provided so subclasses can override with more information; Throwable calls *getMessage()* by the default
- *public synchronized Throwable getCause()* – returns the cause of the exception
- *public String toString()* – returns Throwable string (name of the object and localized message)
- *public void printStackTrace() - prints call stack trace*
  - Output to standard error by default but we can specify *PrintStream* or *PrintWriter* as an argument (overloaded method)

Things to remember about exception handling

- Multiple statements in your method can throw an exception; enclose each statement that might throw an exception into its own try-catch block
  - Multiple exception handlers can be associated with a single *try* block
- Use default exception in addition to specific types of anticipated exceptions
- Each *try* statement can have zero or more *catch* blocks but only a single *finally* block
  - *finally* block is optional
- It is a good idea to clean up resources in *finally* block
- It is not a good practice to re-throw exceptions after you catch them

Best exception handling practices

- Use as specific exceptions as possible
- Throw early in your program
  - Avoids large call stacks
- Close and release resources in *finally* block
- Log exceptions for debugging and record
  - Document/record all exceptions thrown
- Try to make as few catch blocks as possible
  - A single catch block for multiple exceptions