# Exception Handling

try, catch, throw, throws & finally

# Exception Handling

- An exception is an event that occurs during the execution of a program that **disrupts the normal flow** of instructions.
- A Java exception is an **object** that describes an exceptional condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is **created** and **thrown** in the method that caused the error.
- That method may choose to handle the exception **itself**, or **pass** it on.
- Either way, at some point, the exception is **caught** and **processed**.
- The core advantage of exception handling is to **maintain the normal flow** of the instructions.
- We have two categories of Exceptions: Checked exceptions and Unchecked exceptions

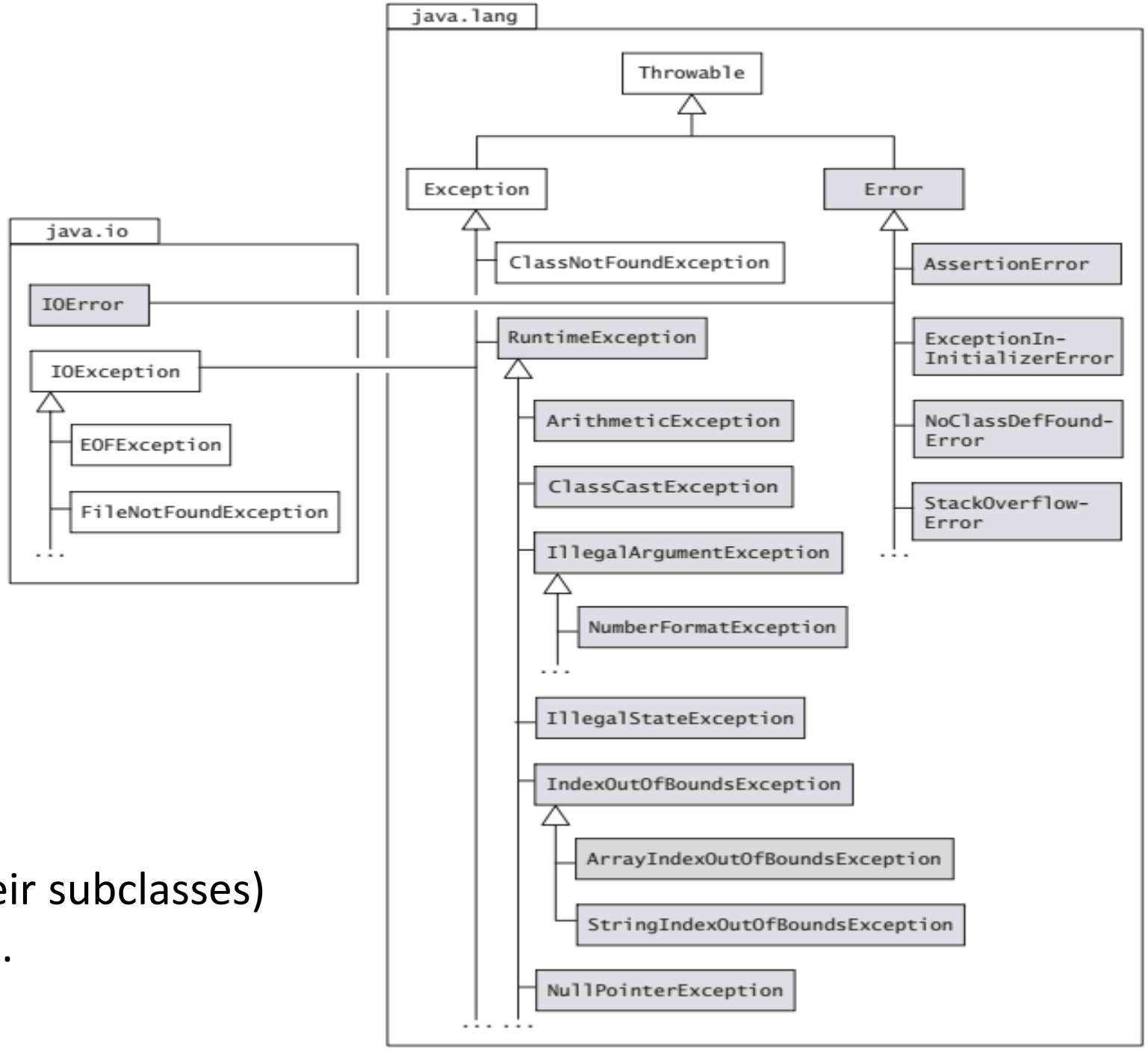
# Checked exceptions

- A checked exception is an exception that is checked by the compiler at compilation-time.
- These are also called as compile time exceptions.
- These exceptions cannot simply be ignored, the programmer should handle these exceptions.

# Unchecked exceptions

- An unchecked exception is an exception that occurs at the time of execution.
- These are also called as Runtime Exceptions.
- These include programming bugs, such as logic errors or improper use of an API.
- Runtime exceptions are ignored at the time of compilation.

# Exception Hierarchy




Classes that are shaded (and their subclasses) represent unchecked exceptions.

# Exception Hierarchy

- All exceptions are derived from the java.lang.Throwable class.
- The two main subclasses Exception and Error constitute the main categories of throwables.
- Except for RuntimeException, Error, and their subclasses, all exceptions are called **checked exceptions**.
- The compiler ensures that if a method can throw a checked exception, directly or indirectly, the method must explicitly deal with it.
- Exceptions defined by Error and RuntimeException classes and their subclasses are known as **unchecked exceptions**,

# Uncaught Exceptions

- Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them.

```
class Demo
{
        public static void main(String[] args) {
                int d = 0, a=0;
                a = 42 / d;
                System.out.println("Value of a ="+a);
        }
}
```

# Uncaught Exceptions

- Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them.

```
class Demo
{
      public static void main(String[] args) {
            int d = 0, a=0;
            a = 42 / d;
            System.out.println("Value of a ="+a);
      }
}
```

**Output:**

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

# Uncaught Exceptions

- When the Java run-time system detects the attempt to divide by zero, it **constructs a new exception object** and then **throws this exception**.
- This causes the execution of the program to stop, because once an exception has been thrown, it must be caught by an **exception handler** and dealt with immediately.
- In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the **default handler** provided by the **Java run-time system**.
- Any exception that is not caught by your program will ultimately be processed by the **default handler**.
- The **default handler** displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

# Using try and catch

- Although the default exception handler provided by the Java run-time system is useful for debugging.
- You will usually want to handle an exception yourself. Doing so provides two benefits.
  - First, it allows you to fix the error.
  - Second, it prevents the program from automatically terminating.
- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try block**.
- Immediately following the **try block**, include a **catch clause** that specifies the exception type that you wish to catch.
- When an exception is encountering in try block, the control is transferred to the catch blocks- if any such blocks are specified.
- On exit from a catch block, normal execution continues unless there is any pending exception that has been thrown and not handled.

# Using try and catch

- Example:

```
class Demo
{
	public static void main(String[] args) {
		int d = 0, a=0;
		try
		{
			a = 42 / d;
		}
		catch(Exception e)
		{
			System.out.println(“Exception caught");
		}
		System.out.println("Value of a ="+a);
	}
}
```

# Using try and catch

- Example:

```
class Demo
{
	public static void main(String[] args) {
		int d = 0, a=0;
		try
		{
			a = 42 / d;
		}
		catch(Exception e)
		{
			System.out.println(“Exception caught");
		}
		System.out.println("Value of a ="+a);
	}
}
```

**Output:**
Exception caught
Value of a =0

# Using try and catch

- Example:

```
class Demo
{
	public static void main(String[] args) {
		int d = 0, a=0;
		try
		{
			a = 42 / d;
			System.out.println("in try block after Exception");
		}
		catch(Exception e)
		{
			System.out.println("Exception caught");
		}
		System.out.println("Value of a ="+a);
	}
}
```

# Using try and catch

- Example:

```
class Demo
{
        public static void main(String[] args) {
                int d = 0, a=0;
                try
                {
                        a = 42 / d;
                        System.out.println("in try block after Exception");
                }
                catch(Exception e)
                {
                        System.out.println("Exception caught");
                }
                System.out.println("Value of a ="+a);
        }
}
```

**Output:**

Exception caught

Value of a =0

# Using try and Multiple catch

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify **two or more catch** clauses, each catching a different type of exception.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed, and execution continues after the try catch block.

# Using try and Multiple catch

- Example:

```
class Demo
{
	public static void main(String[] args) {
		int d = 0, a=0;
		try
		{
			a = 42 / d;
			System.out.println("in try block after Exception");
		}
		catch(Exception e)
		{
			System.out.println("Exception caught");
		}
		catch(Throwable e)
		{
			System.out.println("Exception caught in Throwable");
		}
		System.out.println("Value of a ="+a);
	}
}
```

# Using try and Multiple catch

- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses.
- This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
- Thus, a subclass would never be reached if it came after its superclass.
- Further, in Java, unreachable code is an error.

# 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.

# Nested try Statements

- Example 1:

```
class Demo
{
	public static void main(String[] args) {
		int d = 0, a=0;
		try{
			try{
				a = 42 / d;
				System.out.println("in try block after Exception");
			}
			catch(ArrayIndexOutOfBoundsException e){
			System.out.println("Nested try catch Block");
			}
		}
		catch(ArithmeticException  e){
			System.out.println("Outer try catch Block");
		}
		catch(Exception e){
			System.out.println("Exception caught in Exception");
		}
		System.out.println("Value of a ="+a);
	}
}
```

# Nested try Statements

- Example 1:

```
class Demo
{
	public static void main(String[] args) {
		int d = 0, a=0;
		try{
			try{
				a = 42 / d;
				System.out.println("in try block after Exception");
			}
			catch(ArrayIndexOutOfBoundsException e){
			System.out.println("Nested try catch Block");
			}
		}
		catch(ArithmeticException  e){
			System.out.println("Outer try catch Block");
		}
		catch(Exception e){
			System.out.println("Exception caught in Exception");
		}
		System.out.println("Value of a ="+a);
	}
}
```

**Output:**

Outer try catch Block

Value of a =0

# Nested try Statements

- Example 2:

```
class Demo
{
	public static void main(String[] args) {
		int d = 0, a=0;
		try{
			try{
				a = 42 / d;
				System.out.println("in try block after Exception");
			}
			catch(ArrayIndexOutOfBoundsException e){
			System.out.println("Nested try catch Block");
			}
		}
		catch(NullPointerException  e){
			System.out.println("Outer try catch Block1");
		}
		catch(ArithmeticException  e){
			System.out.println("Outer try catch Block2");
		}
		catch(Exception e){
			System.out.println("Exception caught in Exception3");
		}
		System.out.println("Value of a ="+a);
	}
}
```

# Nested try Statements

- Example 2:

```
class Demo
{
	public static void main(String[] args) {
		int d = 0, a=0;
		try{
			try{
				a = 42 / d;
				System.out.println("in try block after Exception");
			}
			catch(ArrayIndexOutOfBoundsException e){
			System.out.println("Nested try catch Block");
			}
		}
		catch(NullPointerException  e){
			System.out.println("Outer try catch Block1");
		}
		catch(ArithmeticException  e){
			System.out.println("Outer try catch Block2");
		}
		catch(Exception e){
			System.out.println("Exception caught in Exception3");
		}
		System.out.println("Value of a ="+a);
	}
}
```

**Output:**

Outer try catch Block2

Value of a =0

# Nested try Statements

- Example 3:

```
class Demo
{
	static int d = 0, a=0;
	static void show()
	{
		try{
			a = 42 / d;
			System.out.println("in try block after Exception");
		}
		catch(ArrayIndexOutOfBoundsException e){
			System.out.println("Nested try catch Block");
		}
	}
	 public static void main(String[] args) {
		try{
			show();
		}
		catch(NullPointerException  e){
			System.out.println("Outer try catch Block1");
		}
		catch(ArithmeticException  e){
			System.out.println("Outer try catch Block2");
		}
		catch(Exception e){
			System.out.println("Exception caught in Exception3");
		}
		System.out.println("Value of a ="+a);
	}
}
```

# Nested try Statements

- Example 3:

```
class Demo
{
	static int d = 0, a=0;
	static void show()
	{
		try{
			a = 42 / d;
			System.out.println("in try block after Exception");
		}
		catch(ArrayIndexOutOfBoundsException e){
			System.out.println("Nested try catch Block");
		}
	}
	 public static void main(String[] args) {
		try{
			show();
		}
		catch(NullPointerException  e){
			System.out.println("Outer try catch Block1");
		}
		catch(ArithmeticException  e){
			System.out.println("Outer try catch Block2");
		}
		catch(Exception e){
			System.out.println("Exception caught in Exception3");
		}
		System.out.println("Value of a ="+a);
	}
}
```

**Output:**

Outer try catch Block2

Value of a =0

# throw Statement

- So far, we have only been catching exceptions that are thrown by the Java run-time system.
- However, it is possible for your program to throw an exception explicitly, using the throw statement.
- The general form of throw:

**throw** ThrowableInstance;

- Here, ThrowableInstance must be an **object** of type Throwable or a subclass of Throwable.
- We can create an exception using **new** operator.

# throw Statement

- Example 3:

```
class Demo
{
	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);
		}
	}
}
```

# throw Statement

- Example 3:

```
class Demo
{
        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);
                }
        }
}
```

**Output:**

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

# 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.
- You do this 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 the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

- Here, exception-list is a comma-separated list of the exceptions that a method can throw.

# throws Clause

- Example 3:

```
class Demo
{
        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);
        }
}
}
```

# throws Clause

- Example 3:

```
class Demo
{
        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);
        }
}
}
```

**Output:**

inside throwOne

caught java.lang.IllegalAccessException: demo

# finally

- 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 (e.g., to free resources such as files and net connections).
- The finally block will execute whether or not an exception is thrown.

# finally

- Example 3:

```
class Demo
{
        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);
        }
        finally{
                System.out.println(" finally executed");
        }
}
}
```

# finally

- Example 3:

```
class Demo
{
        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);
        }
        finally{
                System.out.println(" finally executed");
        }
}
}
```

**Output:**

inside throwOne
caught java.lang.IllegalAccessException: demo
finally executed

# Built-In Exceptions

- Java defines several exception classes.
- The most general of these exceptions are subclasses of the standard type RuntimeException.
- As previously explained, these exceptions need not be included in any method's **throws** list.
- These are called **unchecked** exceptions because the **compiler** does not check to see if a method handles or throws these exceptions.
- The **checked** exceptions must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself.
- In addition to the exceptions in java.lang, Java defines several more that relate to its other standard packages.

# Built-In Exceptions

Throwable class:

- The Throwable class is the superclass of all **errors** and **exceptions** in the Java language.
- Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.
- Similarly, only this class or one of its subclasses can be the argument type in a catch clause.
- By convention, class Throwable and its subclasses have two constructors, one that takes no arguments and one that takes a String argument that can be used to produce a detail message.

# Built-In Exceptions

Throwable class:

Constructor Summary:

- Throwable()

  Constructs a new throwable with null as its detail message.

- Throwable(String message)

  Constructs a new throwable with the specified detail message.

- Throwable(String message, Throwable cause)

  Constructs a new throwable with the specified detail message and cause.

- Throwable(Throwable cause)

  Constructs a new throwable with the specified cause and a detail message.

# Built-In Exceptions

Throwable class:

Method Summary:

- Throwable fillInStackTrace()

  Fills in the execution stack trace. This method records within this Throwable object information about the current state of the stack frames for the current thread.

- Throwable getCause()

  Returns the cause of this throwable or null if the cause is nonexistent or unknown.

- String getLocalizedMessage()

  Creates a localized description of this throwable.

- String getMessage()

  Returns the detail message string of this throwable.

# Built-In Exceptions

Throwable class:

Method Summary:

- StackTraceElement[] getStackTrace()

  Provides programmatic access to the stack trace information printed by printStackTrace().

- Throwable initCause(Throwable cause)

  Initializes the cause of this throwable to the specified value.

- void printStackTrace()

  Prints this throwable and its backtrace to the standard error stream.

- void printStackTrace(PrintStream s)

  Prints this throwable and its backtrace to the specified print stream.

- void printStackTrace(PrintWriter s)

  Prints this throwable and its backtrace to the specified print writer.

# Built-In Exceptions

Throwable class:

Method Summary:

- void setStackTrace(StackTraceElement[] stackTrace)

  Sets the stack trace elements that will be returned by getStackTrace() and printed by printStackTrace() and related methods.

- String toString()

  Returns a short description of this throwable.

# Built-In Exceptions

Exception class:

- The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.

# Built-In Exceptions

Exception class:

Constructor Summary:

- Exception()

  Constructs a new exception with null as its detail message.

- Exception(String message)

  Constructs a new exception with the specified detail message.

- Exception(String message, Throwable cause)

  Constructs a new exception with the specified detail message and cause.

- Exception(Throwable cause)

  Constructs a new exception with the specified cause and a detail message.

# Built-In Exceptions

RuntimeException class:

- RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.
- A method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught.

# Built-In Exceptions

RuntimeException class:

Constructor Summary:

- RuntimeException()

  Constructs a new runtime exception with null as its detail message.

- RuntimeException(String message)

  Constructs a new runtime exception with the specified detail message.

- RuntimeException(String message, Throwable cause)

  Constructs a new runtime exception with the specified detail message and cause.

- RuntimeException(Throwable cause)

  Constructs a new runtime exception with the specified cause and a detail message.

# Built-In Exceptions (Unchecked)

- ArithmeticException

- Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class.

- ArrayIndexOutOfBoundsException

- Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

- ArrayStoreException

- Thrown to indicate that an attempt has been made to store the wrong type of object into an array of objects. For example, the following code generates an ArrayStoreException:

```
Object x[] = new String[3];
x[0] = new Integer(0);
```

# Built-In Exceptions (Unchecked)

- ClassCastException

- Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance. For example, the following code generates a ClassCastException:

```
Object x = new Integer(0);
System.out.println((String)x);
```

- EnumConstantNotPresentException

- Thrown when an application tries to access an enum constant by name and the enum type contains no constant with the specified name.

- IllegalArgumentException

- Thrown to indicate that a method has been passed an illegal or inappropriate argument.

# Built-In Exceptions (Unchecked)

- IndexOutOfBoundsException
- Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range.
- NegativeArraySizeException
- Thrown if an application tries to create an array with negative size.
- NullPointerException
- Thrown when an application attempts to use null in a case where an object is required. These include:
  - Calling the instance method of a null object.
  - Accessing or modifying the field of a null object.
  - Taking the length of null as if it were an array.
  - Accessing or modifying the slots of null as if it were an array.
  - Throwing null as if it were a Throwable value.
- Applications should throw instances of this class to indicate other illegal uses of the null object.

# Built-In Exceptions (Unchecked)

- NumberFormatException
- Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.
- SecurityException
- Thrown by the security manager to indicate a security violation.
- StringIndexOutOfBoundsException
- Thrown by String methods to indicate that an index is either negative or greater than the size of the string. For some methods such as the charAt method, this exception also is thrown when the index is equal to the size of the string.
- UnsupportedOperationException
- Thrown to indicate that the requested operation is not supported.

# Built-In Exceptions (Checked)

➢ClassNotFoundException

- Thrown when an application tries to load in a class through its string name using:

  The forName method in class Class.
  The findSystemClass method in class ClassLoader .
  The loadClass method in class ClassLoader.

- but no definition for the class with the specified name could be found.

➢CloneNotSupportedException

- Thrown to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface.
- Applications that override the clone method can also throw this exception to indicate that an object could not or should not be cloned.

# Built-In Exceptions (Checked)

- IllegalAccessException

- An IllegalAccessException is thrown when an application tries to reflectively create an instance (other than an array), set or get a field, or invoke a method, but the currently executing method does not have access to the definition of the specified class, field, method or constructor.

- InstantiationException

- Thrown when an application tries to create an instance of a class using the newInstance method in class Class, but the specified class object cannot be instantiated. The instantiation can fail for a variety of reasons including but not limited to:
  - the class object represents an abstract class, an interface, an array class, a primitive type, or void
  - the class has no nullary constructor

# Built-In Exceptions (Checked)

- InterruptedException
- Thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity.
- NoSuchFieldException
- Signals that the class doesn't have a field of a specified name.
- NoSuchMethodException
- Thrown when a particular method cannot be found.

# Creating Your Own Exception

- Although Java's built-in exceptions handle most common errors.
- However, you will probably want to create your own exception types to handle situations specific to your applications.
- To do just define a subclass of Exception.
- The Exception class does not define any methods of its own. It does, of course, inherit those methods provided by Throwable.
- Exception defines four public constructors.

# Creating Your Own 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);
                }
        }
}
```

# Creating Your Own 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);
                }
        }
}
```

**Output:**

Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]