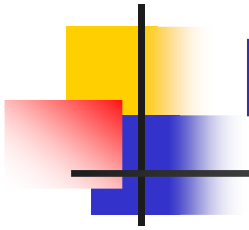
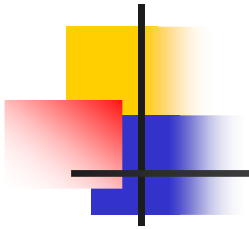
Exception Handling

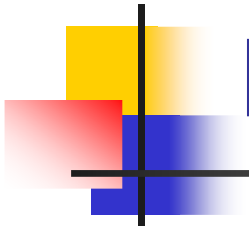


Exception-Handling Fundamentals

- An *exception* is an abnormal condition that arises in a code sequence at run time
- 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

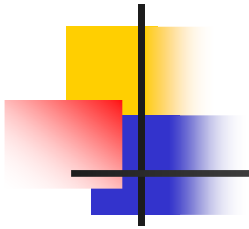


- An exception can be caught to handle it or pass it on
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code



Exception-Handling Fundamentals

- Java exception handling is managed by via five keywords: **try, catch, throw, throws,** and **finally**
- Program statements to monitor are contained within a **try** block
- If an exception occurs within the **try** block, it is thrown
- Code within **catch** block catch the exception and handle it

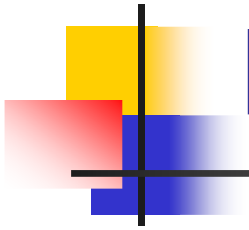


- System generated exceptions are automatically thrown by the Java run-time system
- To manually throw an exception, use the keyword **throw**
- Any exception that is thrown out of a method must be specified as such by a **throws** clause
- Any code that absolutely must be executed before a method returns is put in a **finally** block

Exception-Handling Fundamentals

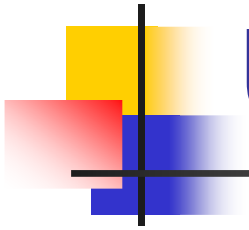
- General form of an exception-handling block

```
try{
    // block of code to monitor for errors
}
catch (ExceptionType1 exOb){
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb){
    // exception handler for ExceptionType2
}
//…
finally{
    // block of code to be executed before try block ends
}
```



Exception Types

- All exception types are subclasses of the built-in class **Throwable**
- Throwable has two subclasses, they are
  - Exception (to handle exceptional conditions that user programs should catch)
    - An important subclass of Exception is **RuntimeException**, that includes division by zero and invalid array indexing
  - Error (to handle exceptional conditions that are not expected to be caught under normal circumstances). i.e. stack overflow



Uncaught Exceptions

- If an exception is not caught by user program, then execution of the program stops and it is caught by the default handler provided by the Java run-time system

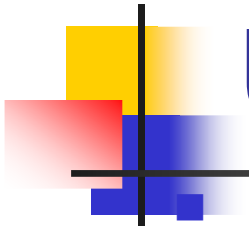
- Default handler prints a stack trace from the point at which the exception occurred, and terminates the program

Ex:

```
class Exc0 {
  public static void main(String args[]) {
    int d = 0;
    int a = 42 / d;
  }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
    at Exc0.main(Exc0.java:4)

Exception in thread "main"
```



Using try and catch

- Handling an exception has two benefits,
  - It allows you to fix the error
  - It prevents the program from automatically terminating
- The **catch** clause should follow immediately the **try** block
- Once an exception is thrown, program control transfer out of the **try** block into the catch block
- Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism

Example

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
      d = 0;
      a = 42 / d;
      System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
      System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```

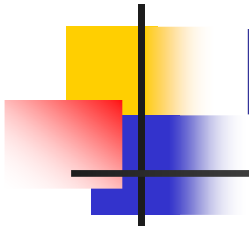
Output:

Division by zero.

After catch statement.

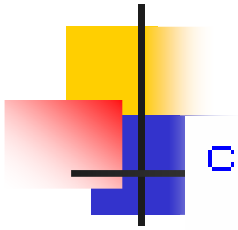
Using try and catch

```
import java.util.Random;

class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();

    for(int i=0; i<10; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Division by zero.");
        a = 0; // set a to zero and continue
      }
      System.out.println("a: " + a);
    }
  }
}
```



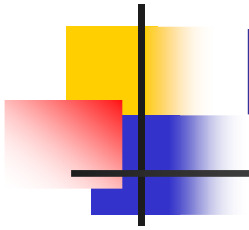
Multiple catch Clauses

- If more than one can occur, then we use multiple catch clauses
- 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



Example

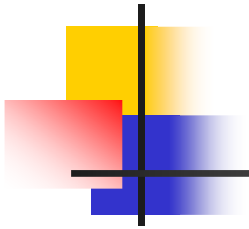
```
class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```



Example (Cont.)

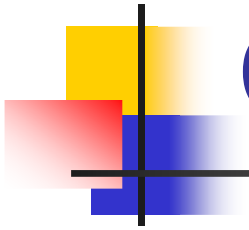
- If no command line argument is provided, then you will see the following output:

```
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks
```



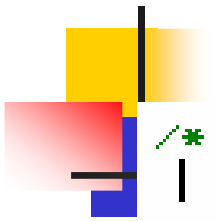
- If any command line argument is provided, then you will see the following output:

```
a = 1
Array index oob:
   java.lang.ArrayIndexOutOfBoundsException
After try/catch blocks.
```



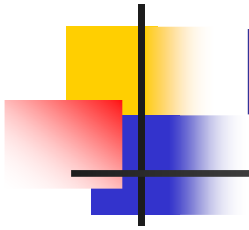
Caution

- Remember that, exception subclass must come before any of of their superclasses
- Because, a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. So, the subclass would never be reached if it come after its superclass
- For example, **ArithmeticException** is a subclass of **Exception**
- Moreover, unreachable code in Java generates error



Exampnle

```
/*  This program contains an error.

    A subclass must come before its superclass in
    a series of catch statements. If not,
    unreachable code will be created and a
    compile-time error will result.
*/
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    } catch(Exception e) {
      System.out.println("Generic Exception catch.");
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
    catch(ArithmeticException e) { // ERROR - unreachable
      System.out.println("This is never reached.");
    }
  }
}
```

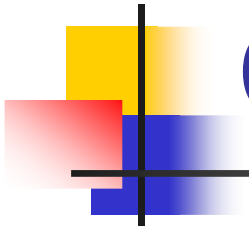


Nested try Statements

- 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, then the next **try** statement's catch handlers are inspected for a match
- If a method call within a **try** block has **try** block within it, then then it is still nested **try**

Example

```
// An example nested try statements.
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try { // nested try block
        /* If one command line arg is used,
           then an divide-by-zero exception
           will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

        /* If two command line args are used
           then generate an out-of-bounds exception. */
        if(a==2) {
          int c[] = { 1 };
          c[42] = 99; // generate an out-of-bounds exception
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
      }

    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```



Output

- When no parameter is given:

  Divide by 0: java.lang.ArithmeticException: / by zero

- When one parameter is given

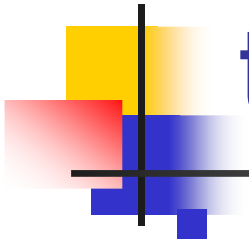
  a = 1

  Divide by 0: java.lang.ArithmeticException: / by zero

- When two parameters are given

  a = 2

  Array index out-of-bounds:
  java.lang.ArrayIndexOutOfBoundsException

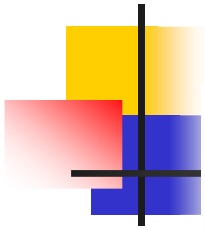


throw

- It is possible for your program to throw an exception explicitly

  throw *TrrowableInstance*

- Here, *TrrowableInstance* must be an object of type **Throwable** or a subclass **Throwable**
- There are two ways to obtain a **Throwable** objects:
  - Using a parameter into a catch clause
  - Creating one with the **new** operator



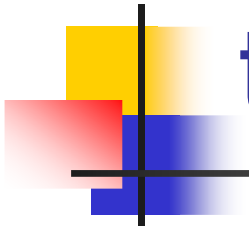
Example

```
// Demonstrate throw.
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // re-throw 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

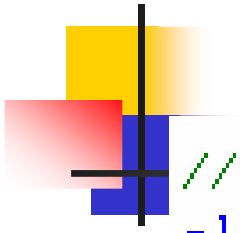


throws

- 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

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

- It is not applicable for **Error** or **RuntimeException,** or any of their subclasses



Example: incorrect program

```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

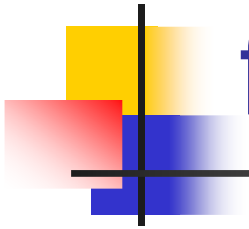
Example: corrected version

```
// This is now correct.
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);
    }
  }
}
```

Output:

Inside throwOne.

Caught java.lang.IllegalAccessException: demo

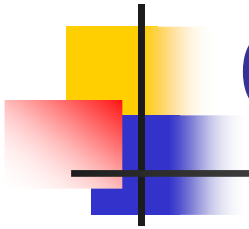


finally

- It is used to handle premature execution of a method (i.e. a method open a file upon entry and closes it upon exit)
- **finally** creates a block of code that will be executed after **try/catch** block has completed and before the code following the **try/catch** block
- **finally** clause will execute whether or not an exception is thrown

Example

```
// Demonstrate finally.
class FinallyDemo {
  // Through an exception out of the method.
  static void procA() {
    try {
      System.out.println("inside procA");
      throw new RuntimeException("demo");
    } finally {
      System.out.println("procA's finally");
    }
  }

  // Return from within a try block.
  static void procB() {
    try {
      System.out.println("inside procB");
      return;
    } finally {
      System.out.println("procB's finally");
    }
  }

  // Execute a try block normally.
  static void procC() {
    try {
      System.out.println("inside procC");
    } finally {
      System.out.println("procC's finally");
    }
  }

  public static void main(String args[]) {
    try {
      procA();
    } catch (Exception e) {
      System.out.println("Exception caught");
    }
    procB();
    procC();
  }
}
```



Output

inside procA

procA's finally

Exception caught

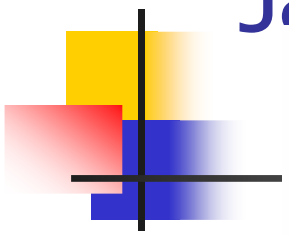
inside procB

procB's finally

inside procC

procC's finally

Java's Built-in Errors



- class java.lang.**Throwable** (implements java.io.Serializable)
  - class java.lang.**Error**
    - class java.lang.**LinkageError**
      - class java.lang.**ClassCircularityError**
      - class java.lang.**ClassFormatError**
        - class java.lang.**UnsupportedClassVersionError**
      - class java.lang.**ExceptionInInitializerError**
      - class java.lang.**IncompatibleClassChangeError**
        - class java.lang.**AbstractMethodError**
        - class java.lang.**IllegalAccessError**
        - class java.lang.**InstantiationError**
        - class java.lang.**NoSuchFieldError**
        - class java.lang.**NoSuchMethodError**
      - class java.lang.**NoClassDefFoundError**
      - class java.lang.**UnsatisfiedLinkError**
      - class java.lang.**VerifyError**
    - class java.lang.**ThreadDeath**
    - class java.lang.**VirtualMachineError**
      - class java.lang.**InternalError**
      - class java.lang.**OutOfMemoryError**
      - class java.lang.**StackOverflowError**
      - class java.lang.**UnknownError**

- **Small letter indicate package name**
- **Capital letter indicate class name**

Java's Built-in Exceptions

- class java.lang.**Throwable** (implements java.io.Serializable)
  - class java.lang.**Exception**
    - class java.lang.**ClassNotFoundException**
    - class java.lang.**CloneNotSupportedException**
    - class java.lang.**IllegalAccessException**
    - class java.lang.**InstantiationException**
    - class java.lang.**InterruptedException**
    - class java.lang.**NoSuchFieldException**
    - class java.lang.**NoSuchMethodException**
    - class java.lang.**RuntimeException**
      - class java.lang.**ArithmeticException**
      - class java.lang.**ArrayStoreException**
      - class java.lang.**ClassCastException**
      - class java.lang.**IllegalArgumentException**
        - class java.lang.**IllegalThreadStateException**
        - class java.lang.**NumberFormatException**
      - class java.lang.**IllegalMonitorStateException**
      - class java.lang.**IllegalStateException**
      - class java.lang.**IndexOutOfBoundsException**
        - class java.lang.**ArrayIndexOutOfBoundsException**
        - class java.lang.**StringIndexOutOfBoundsException**
      - class java.lang.**NegativeArraySizeException**
      - class java.lang.**NullPointerException**
      - class java.lang.**SecurityException**
      - class java.lang.**UnsupportedOperationException**