Robustness & Exception Handling

Robust Programs

- A robust program deals gracefully with unexpected input (among other things).
- How can we make this more robust?

More specifically,

what happens if the user doesn't enter an integer, [while program is waiting for that], instead the user press any character input?

Or,

if someone access an array beyond the allocation of array.

Exceptions

- An exception is an abnormal condition that arises in a piece of code at run time
  - An exception is a run-time error
- If these exceptions are not prevented or at least handled properly,
  - either the program will be aborted abnormally,
  - or the incorrect result will be carried on.

Exception Handling Mechanism

- When an exception condition arises,
  - an object of the respective exception class is created and thrown in the method that caused the exception.
  - That method may choose to handle the exception itself or pass it on.
  - Either way, at some point, the exception is caught and processed

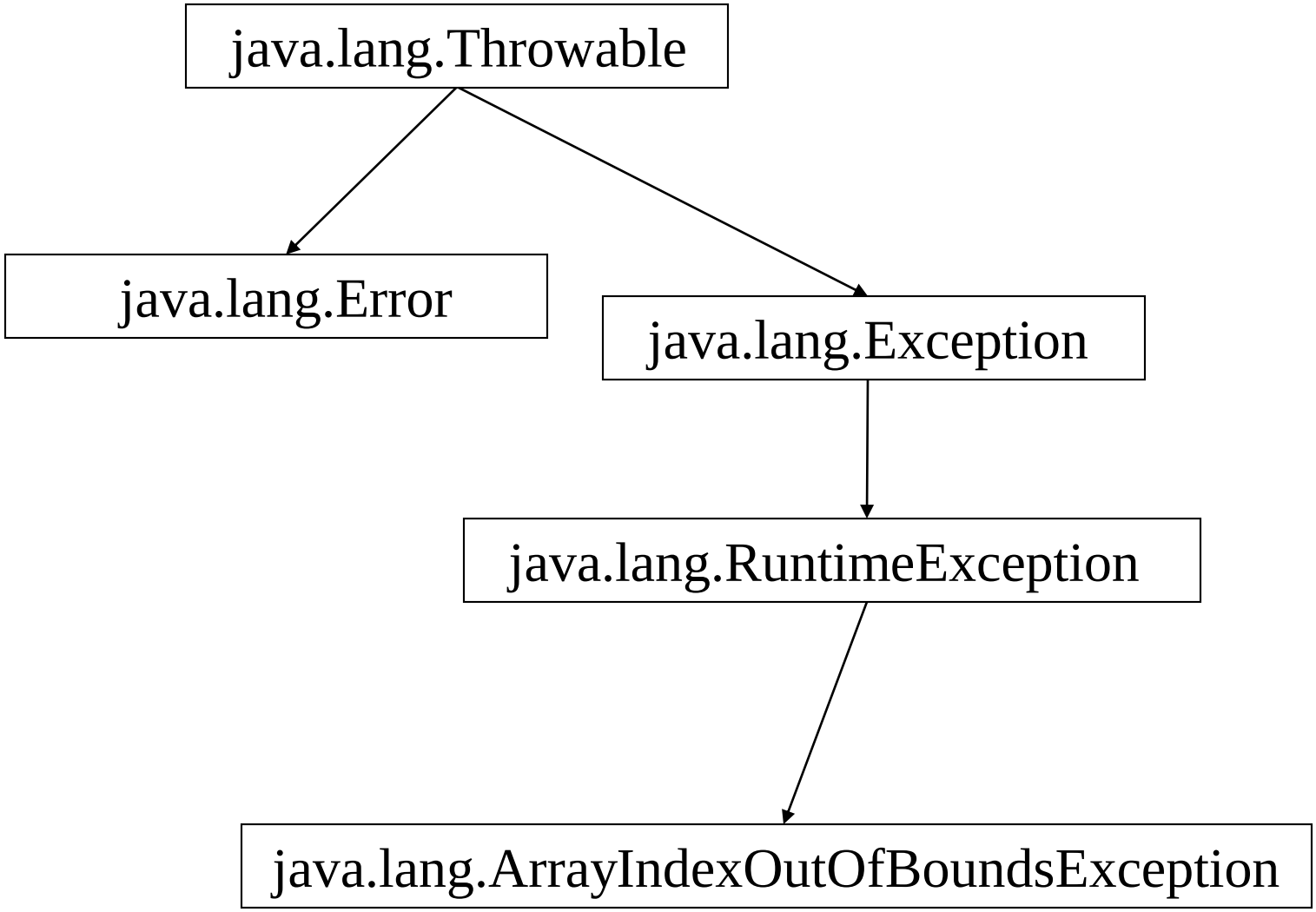
- Exceptions can be generated by Java run-time system or can be manually generated by your code
  - Java exception handling is managed via five keywords: try, catch, throw, throws, finally

Exception Handling Mechanism

- Program statements that you want to monitor for exceptions are contained within a **try** block.
  - If exception occurs within **try** block, it is thrown.
- Your code can catch this exception using **catch** and handle it.
- System generated exceptions are *automatically* thrown by Java run-time system
  - To manually throw an exception, use the keyword **throw**

Exception Handling Mechanism

- 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
- Any code that must be executed before a method returns is put in a **finally** block


java.lang.Throwable
java.lang.Error
java.lang.Exception
java.lang.RuntimeException
java.lang.ArrayIndexOutOfBoundsException

Hierarchy of Exceptions

- The class **Exception** is used for exceptional conditions that user programs can catch

- The class **Error**

  - defines the conditions that should not be expected to be caught under normal circumstances.

  - responsible for giving errors in some catastrophic failures that can't usually be handled by your program.

- The class **RuntimeException** is used for exceptions
  - that are automatically defined for the programs during run time in response to some execution error.

Exceptions

- Some examples of exceptions are:
  - IndexOutOfBoundsException
  - NullPointerException
  - NumberFormatException
  - ArithmeticException
  - FileIOException

Java’s Built-in Exceptions

- Some Runtime Exception Subclasses
  - ArithmeticException
  - ArrayIndexOutofBoundException
  - ClassCastException
  - NullPointerException
  - NumberFormatException
- Some Exception Subclasses
  - ClassNotFoundException
  - NoSuchMethodException
  - IllegalAccessException

Errors

- Some Error Subclasses
  - OutofMemoryError
  - VirtualMachineError
  - StackOverflowError

```
try {
  // some code that might throw an exception
}
catch (ExceptionType1 excepObj) {
  // exception handler for ExceptionType1
}
catch (ExceptionType2 excepObj) {
  // exception handler for ExceptionType2
}
//…
finally {
// code to be executed before try block ends
}
```

Here, ExceptionType1 and ExceptionType2 is the type of exception that has occured

Uncaught Exceptions

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

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

Uncaught Exceptions

```
class Exc1 {
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String args[]) {
    Exc1.subroutine();
  }
}
```

```
Java.lang.ArithmaticException: / by zero
        at Exc1.subroutine (Exc1.java:4)
        at Exc1.main(Exc1.java:7)
```

‘try’ and ‘catch’

```
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.");
  }
}
```

```
Division by zero
After catch statement
```

Multiple catch Clauses

```
class MultiCatch{
 public static void main(String args[]){
  try {
    int a = args.legth;
    System.out.println("a = "+a);
    int b = 42/a;
    int c[] = {1};
    c[2] = 99;
  } catch (ArithmeticException e) {
   System.out.println("Divide by 0: "+e);
  } catch (ArrayIndexOutOfBoundsException e) {
   System.out.println("Array index oob: "+e);
  } finally {
    System.out.println("I am in finally block!");
  }
}
```

MultiCatch

- java MultiCatch

```
a = 0
Divide by 0:
  java.lang.ArithmeticException: / by zero
I am in finally block!
```

- java MultiCatch TestArg

```
a = 1
Array index oob:
  java.lang.ArrayIndexOutOfBoundsException
I am in finally block!
```

Multiple catch Clauses (**restriction**)

```
/*  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.");
    }
  }
}
```

// 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 in 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);  }
  }
}
```

```
C:\ java NestTry
Divide by 0: java.lang.ArithmaticException: / by zero

C:\ java NestTry One
a = 1
Divide by 0: java.lang.ArithmaticException: / by zero

C:\ java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException: 42
```

throw

- General form of **throw**:

throw thrObj;

Where ‘thrObj’ must be an object of type **Throwable** or a subclass of **Throwable**

  - Simple types, such as **int** or **char**, as well as non-**Throwable** classes, [such as **String** and **Object**], cannot be used as exceptions.

- The flow of execution stops immediately after the **throw** statement;
  - any subsequent statements are not executed

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

```
Caught in demoproc

Recaught:
java.lang.NullPointerException: demo
```

throw

- Here is the output

```
Caught in demoproc
Recaught: java.lang.NullPointerException:
  demo
```

- All of Java's built-in run-time exceptions have two constructors:
  - one with no parameter
  - one takes a string parameter
  - When the second form is used, the argument specifies a string that describes the exception
    - This string is displayed when the object is used as an argument to print() or println()

Throwing Exceptions

- Some exceptions are thrown "automatically" by the Java Virtual Machine.

  e.g. IndexOutOfBoundsException

- You can also throw some exceptions yourself.

The **throws** clause

- When do you need a throws?
- There are two types of exceptions in Java
  - checked exceptions, and
  - unchecked exceptions.

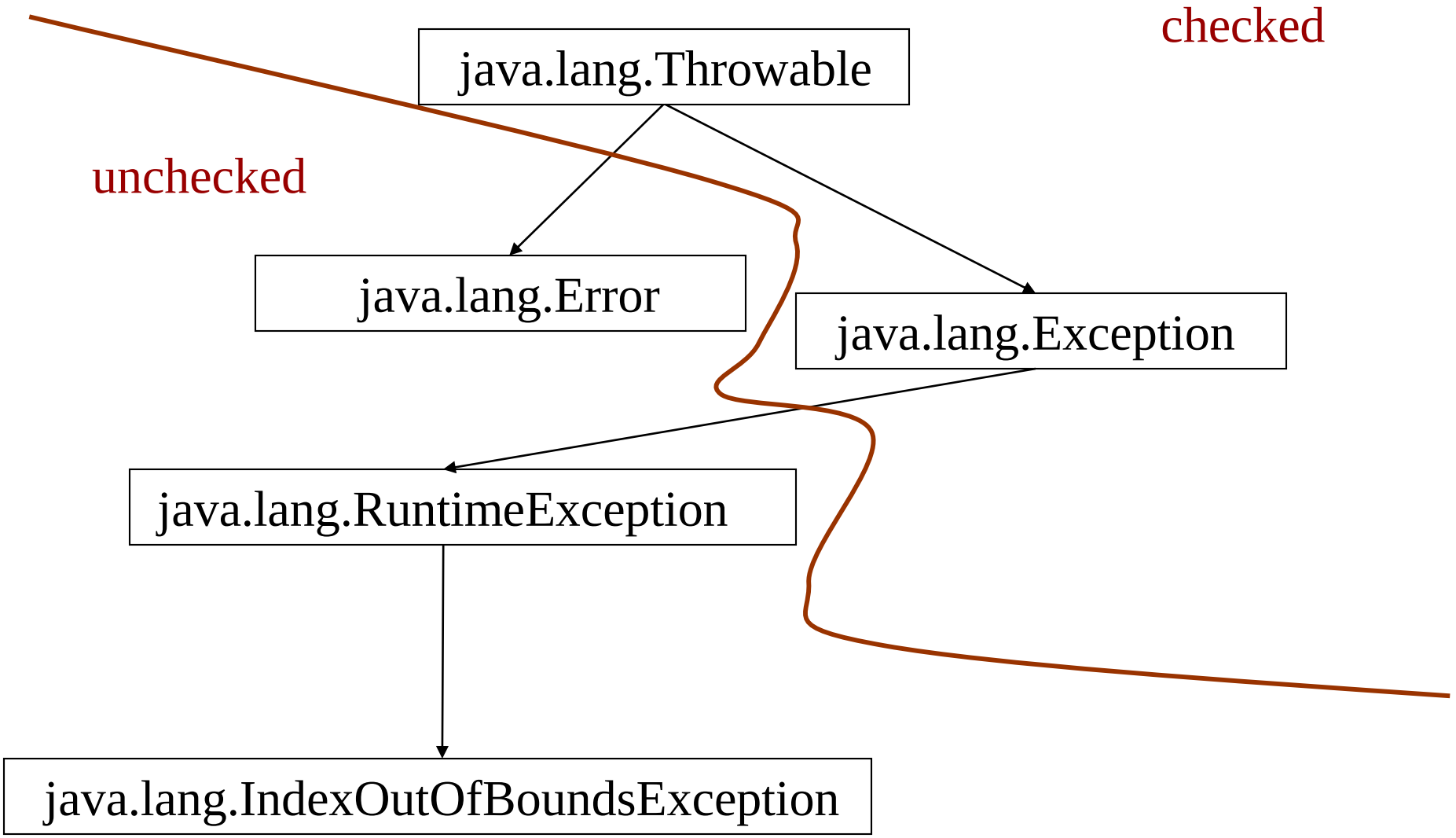
- Checked exceptions require a throws clause whenever they might be thrown.
- Unchecked exceptions are things like NullPointerException, and IndexOutOfBoundsException. needs no throws clause.

Checked vs Unchecked

- *Unchecked* exceptions are exceptions that are instances of –

  java.lang.RuntimeException, java.lang.Error, or one of their subclasses.

  – They are called **unchecked exceptions** because the compiler does not check to see if a method handles or throws these exceptions.
- Everything else is a *checked* exception.


checked
java.lang.Throwable
unchecked
java.lang.Error
java.lang.Exception
java.lang.RuntimeException
java.lang.IndexOutOfBoundsException

The throws clause

- 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
- All other exceptions that a method can throw must be declared in the **throws** clause.
  - Otherwise, a compile time error will result
- General form

```
type methodName(paramList) throws exceptionList
{ // body of method}
```

```
// 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();
  }
}
```

```
class ThrowsDemo{
    static void throwOne() throws IllegalAccessException{
            System.out.println("In throwOne");
            throw new IllegalAccessException("demo");
 }

 public static void main(String args[]){
    try {
            throwOne();
    } catch (IllegalAccessException e) {
            System.out.println("Caught: "+e);
    }
  }
}
```

```
In throwOne
Caught: java.lang.IllegalAccessException: demo
```

Finally

- The **finally** clause can be useful for closing file handles and freeing up any other resources
  - that might have been allocated at the beginning of a method with the intent of disposing of them before returning
- **finally** creates a block of code that will be executed after try/catch has completed.
  - It will executed whether or not an exception is thrown.
- The **finally** clause is optional
- Each **try** statement requires at least one **catch** or a **finally** clause

```
// 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();
  }
}
```

inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally

Creating Own Exception

```
// This program creates a custom exception type.
class MyException extends Exception {
  private int detail;

  MyException(int a) {
    detail = a;
  }

  public String toString() {
    return "MyException[" + detail + "]";
  }
}
```

Creating Own Exception

```
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);
    }
  }
}
```

Called compute (1)

Normal exit

Called compute (20)

Caught: MyException [20]