Object-Oriented Programming: Application to the Java Language

Exceptions

Exception

- Definition
  - An exception is a signal indicating that something exceptional (such as an error) has occurred.
  - It interrupts the normal execution flow of the program.

- Purpose
  - Handling errors is essential: poor error handling can lead to catastrophic consequences (e.g., Ariane 5).
  - The mechanism is simple and easy to read.
    - It allows grouping code specifically for error handling.
    - It enables the possibility of "recovering" from an error at various levels of an application (propagation through the method call stack).

- Vocabulary
  - **Throwing** an exception means signaling an error.
  - **Catching** an exception allows handling the error.

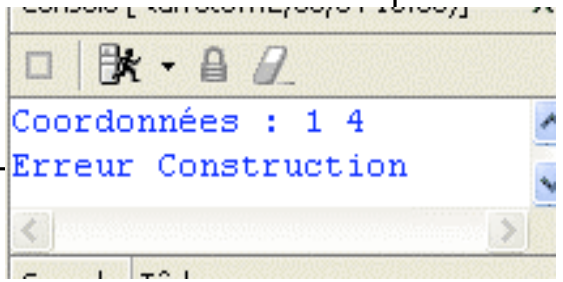
Exception

- Example: throwing and catching an exception

```
public class Point {
     ... // Declaration of attributes

     ... // Other methods and builders

     public Point(int x, int y) throws ErrConst {
          if ((x < 0) || (y < 0)) throw new ErrConst();
          this.x = x ; this.y = y;
     }

     public void affiche() {
          System.out.println("Coordinates: " + x + " " + y);
     }
}
```

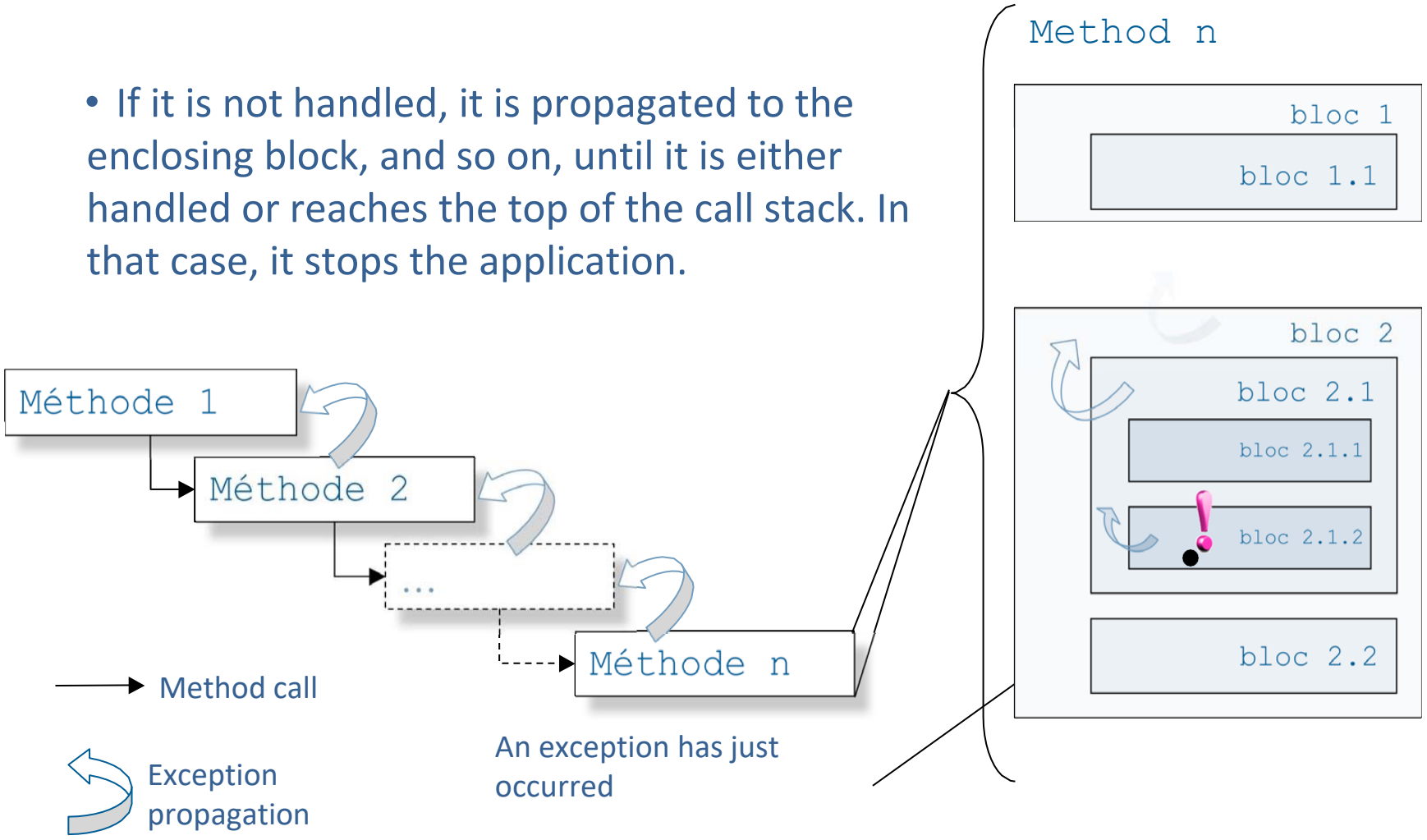
```
public class Test {
     public static void main(String[] argv) {
          try {
               Point a = new
               Point(1,4); a.affiche();
               a = new Point(-2,
               4); a.affiche();
          } catch (ErrConst e) {
               System.out.println("Construction error");
               System.exit(-1);
          }
     }
}
```

Note: The *ErrConst* class is not yet defined. This will be addressed later



Exception: mechanism

- Explanation
  - When an exceptional situation is encountered, an exception is thrown.
  - If it is not handled, it is propagated to the enclosing block, and so on, until it is either handled or reaches the top of the call stack. In that case, it stops the application.



Exception: Throwing or Triggering

- A method declares that it can throw an exception using the **throws** keyword

```
public Point(int x, int y) throws ErrConst {
    ...
}
```

Allows the *Point* constructor to throw an *ErrConst* exception

- The method throws an exception by creating a new exception object using the **throw** keyword

```
public Point(int x, int y) throws ErrConst {
    if ((x < 0) || (y < 0)) throw new ErrConst();
    this.x = x ; this.y = y;
}
```

Creation of a new exception object

- The method calls code that throws an exception

```
public Point(int x, int y) throws ErrConst {
    checkXYValue(x,y);
    this.x = x ; this.y = y;
}
```

```
private void checkXYValue(in x, int y)
throws ErrConst {
    if ((x < 0) || (y < 0))
        throw new ErrConst();
}
```

Exception: catching or handling

- Here, we are talking about an exception handler. It involves taking actions to handle the exceptional situation.
- We enclose a set of instructions that might trigger an exception within a **try {...}** block.

```
try {
    Point a = new Point(1,4);
    a.affiche();
    a = new Point(-2, 4);
    a.affiche();
}
```

Risky methods. They are "monitored."

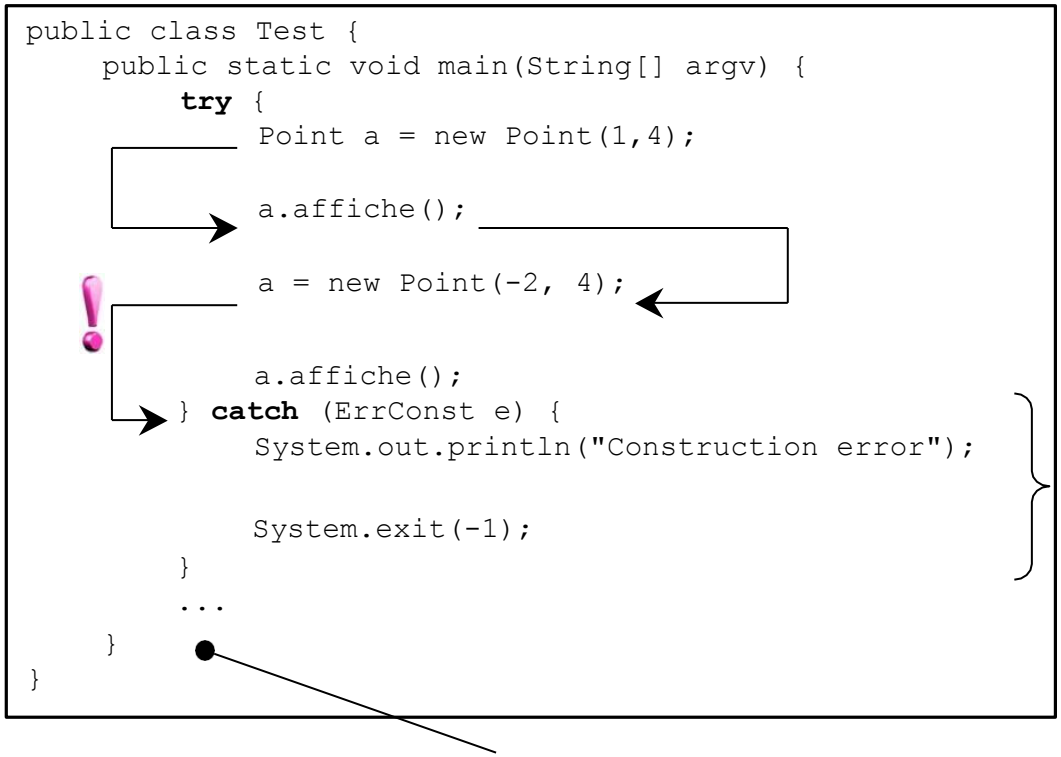
- Risk management is achieved through blocks **catch(TypeException e) {...}**

```
} catch (ErrConst e) {
    System.out.println("Construction error");
    System.exit(-1);
}
```

- These blocks allow capturing exceptions of the specified type and executing appropriate actions.

Exception: catching or handling

- Understanding the mechanism of capturing exceptions



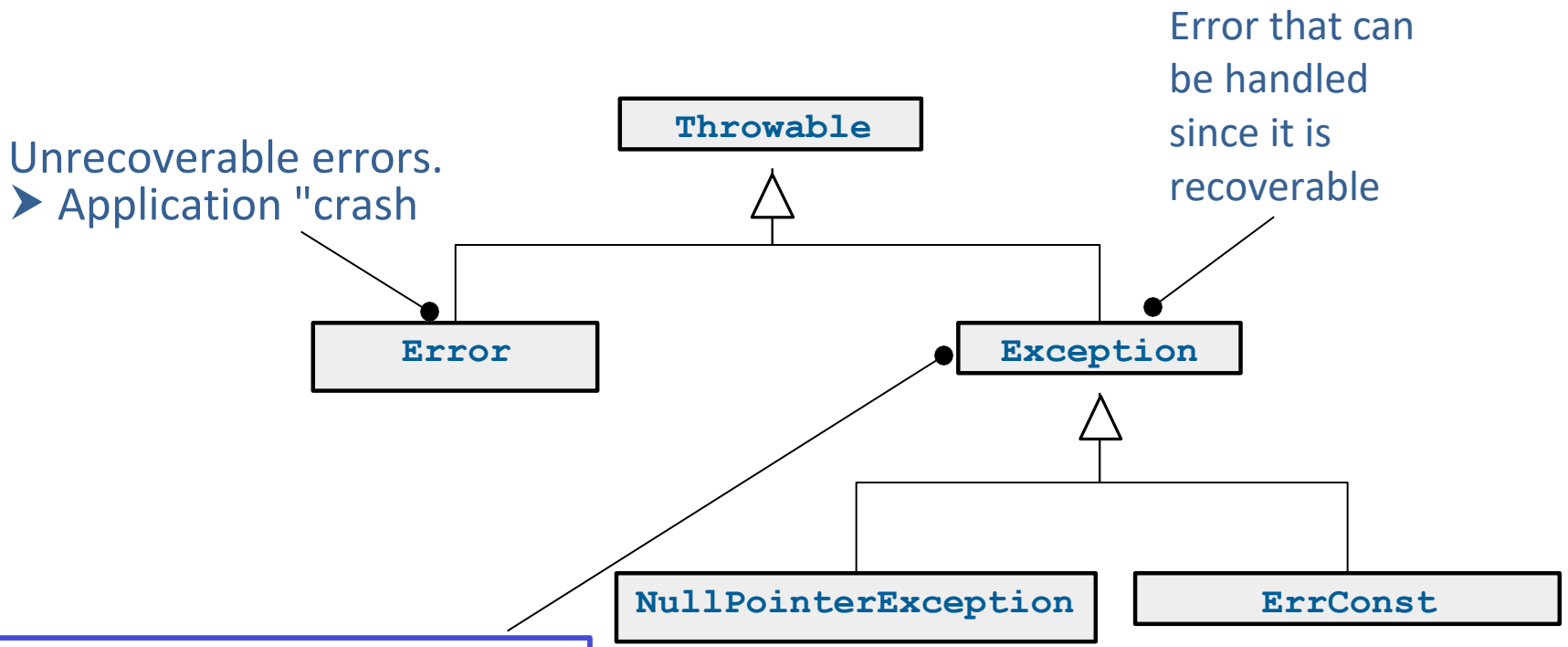
The exceptional error is handled by the **catch** block.

Then, execution continues outside the **try-catch** block.

Note: if an error occurs, the program stops (System.exit(-1)).

Exception: modeling

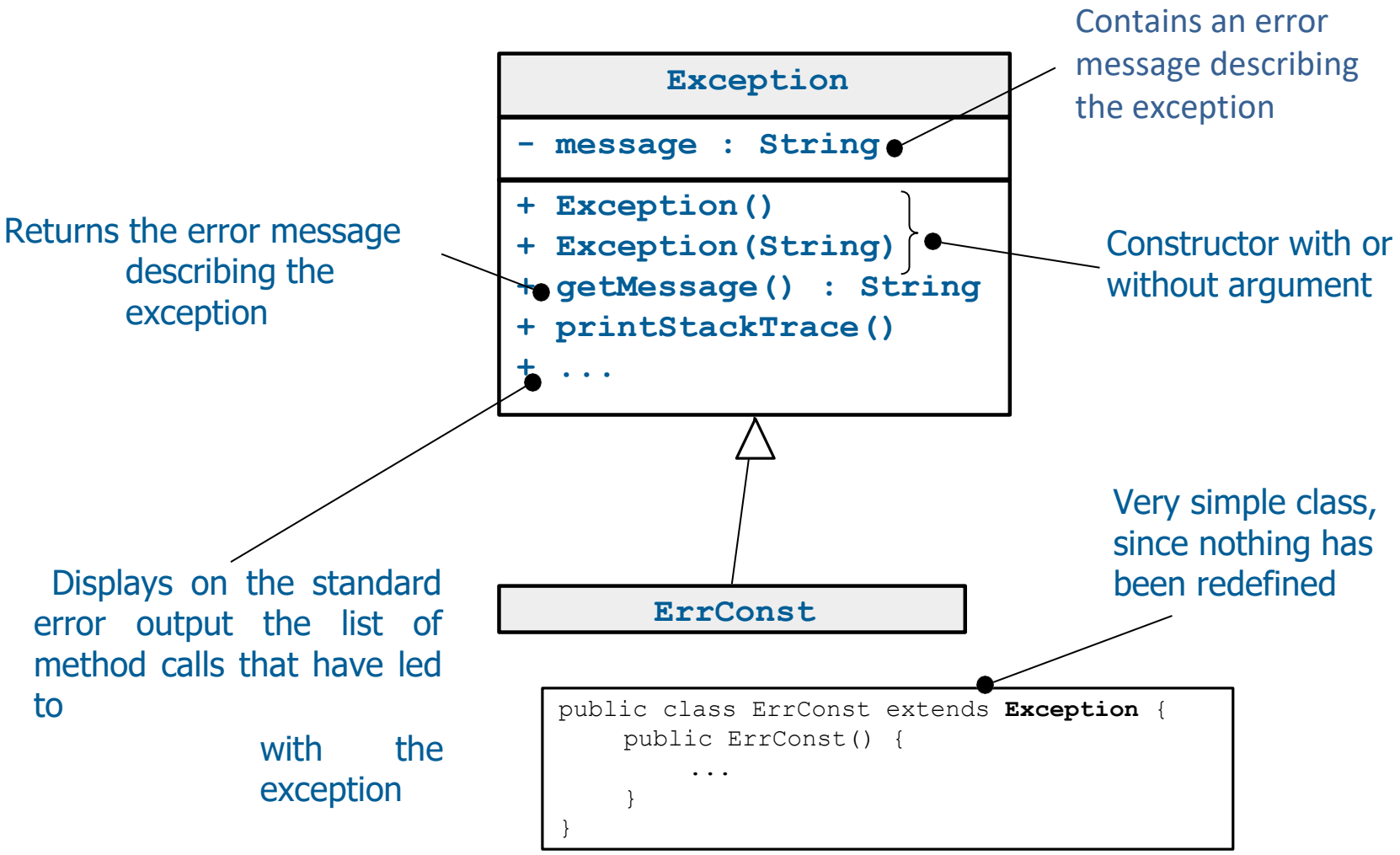
- Exceptions in Java are considered as objects
- Every exception must be an instance of a subclass of the class *java.lang.Throwable*



To define new exception types, the *Exception* class should be subclassed

Exception: modeling

- Exceptions are objects, so we can define : Specific attribute and Methods



Exception: modeling

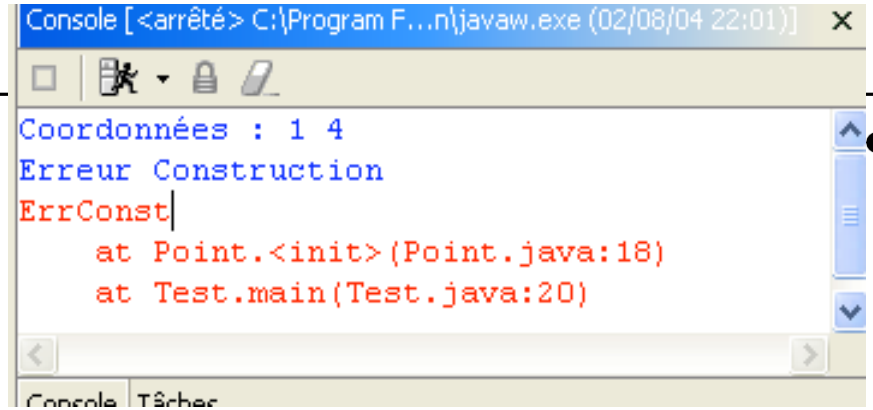
- Example: using the *ErrConst* object

Error type *ErrConst*
which inherits from Exception

```
public class Test {
    public static void main(String[] argv) {
        try {
            ...
        } catch (ErrConst e) {
            System.out.println("Construction error");

            System.out.println(e.getMessage());

            e.printStackTrace();

            System.exit(-1);
        }
        ...
    }
}
```

Error display

Displaying the list of methods



Exception: catching all ...

- It is possible to capture more than one exception. A **try** block and several **catch** blocks

```
public class Point {
    public void deplace(int dx, int dy) throws ErrDepl {
        if (((x+dx) < 0) || ((y+dy) <0)) throw new ErrDepl();
            x += dx ; y +=dy;
    }

    public Point(int x, int y) throws ErrConst {
        if ((x < 0) || (y < 0)) throw new ErrConst();
        this.x = x ; this.y = y;
    }
}
```

Defining a new method that throws an exception

```
public class Test {
    public static void main(String[] argv) {
        try {
            ... // Block in which to detect ErrConst and
                    ErrDepl exceptions
        } catch (ErrConst e) {
            System.out.println("Construction error");
            System.exit(-1);
        } catch (ErrDepl e) {
            System.out.println("Displacement error");
            System.exit(-1);
        }
    }
}
```

Catch the new exception of type *ErrDepl*

Exception: catch them all ...

- Any method likely to raise an exception must
  - Either catch it (***try catch*** block)
  - Either explicitly declare that it can throw an exception (keyword ***throws***)
- Exceptions declared in the ***throws*** clause of a method are ...

Exceptions raised in the (*Point*) method and not caught by it

```
public Point(int x, int y) throws ErrConst {
     if ((x < 0) || (y < 0)) throw new
     ErrConst(); this.x = x ; this.y = y;
}
```

Exceptions lifted in methods (*checkXYValue*) called by the (*Point*) method and not caught by it

```
public Point(int x, int y) throws
     ErrConst { checkXYValue(x,y);
     this.x = x ; this.y = y;
}
```

```
private void checkXYValue(in x, int y)
throws
ErrConst {
      if ((x < 0) || (y < 0))
          throw new ErrConst();
}
```

Exception: catch them all ...

- Make sure exceptions are under control

```
public class Point {
     public void deplace(int dx, int dy) throws ErrDepl {
          if (((x+dx) < 0) || ((y+dy) <0)) throw new ErrDepl();
               x += dx ; y +=dy;
     }

     public void transform() {
          ...
          this.deplace(...);
     }
}
```

```
public class ErrDepl extends Exception {
     public ErrDepl() {
          ...
     }
}
```

Don't forget to handle an exception, otherwise the compiler won't miss you!!!!

Exception: catch them all ...

- There are two ways to ensure correct compilation

```
public class Point {
     public void deplace(int dx, int dy) throws ErrDepl {
          if (((x+dx) < 0) || ((y+dy) <0)) throw new ErrDepl();
               x += dx ; y +=dy;
     }

     public void transform() {
          ...
          this.deplace(...);
     }
}
```

Either by explicitly adding the **throws** instruction to the *transform* method, so as to redirect the error

```
public void transform()
     throws ErrDepl {
     ...
     this.deplace(...);
}
```

Either surround a **try ... catch** block with the method that may cause problems

```
public void transform() {
     try {
          ...
          this.deplace(...);
     } catch (ErrDepl e) {
          e.printStackTrace();
     }
}
```

Exception: transmission of information

- Possibility of enriching the *ErrConst* class by adding attributes and methods to enable communication

```
public class Point {
    public Point(int x, int y) throws ErrConst {
        if ((x < 0) || (y < 0)) throw new ErrConst(x,y);
        this.x = x ; this.y = y;
    }
    ...
}
```

```
public class ErrConst extends Exception {
    private int abs, ord;

    public ErrConst(int x, int y) {
        this.abs = x;
        this.ord = y;
    }

    public int getAbs() { return this.abs; }
    public int getOrd() { return this.ord; }
}
```

ErrConst
- abs, ord : int
+ ErrConst(x,y) + getAbs : int + getOrd : int

```
public class Test {
    public static void main(String[] argv) {
        try {
            ...
            a = new Point(-2, 4);
        } catch (ErrConst e) {
            System.out.println("Construction point error");
            System.out.println("Desired coordinates: "
              + e.getAbs() + " " + e.getOrd());
            System.exit(-1);
        }...
```

ErrConst displays the values that caused *Point* construction to fail

Exception: finally

- *Finally* block: this is an optional instruction that can be used for "clean-up".
- It is executed regardless of the result of the *try* block (i.e. whether or not it has triggered an exception).
- Allows you to specify code whose execution is guaranteed whatever happens
- The benefits are twofold
  - Gather in a single block a set of instructions that would otherwise have to be duplicated
  - Perform processing after the *try* block, even if an exception has been raised and not caught by the *catch* blocks.

Exception: finally

- Example: terminate correctly with *finally*

```
public class Test {
    public static void main(String[] argv) {
        try {
            ... // Block in which to detect ErrConst and
                    ErrDepl exceptions
        } catch (ErrConst e) {
            System.out.println("Construction error");
            System.out.println("End of program");
            System.exit(-1);
        } catch (ErrDepl e) {
            System.out.println("Displacement error");
            System.out.println("End of program");

            System.exit(-1);
        }
    }
}
```

These instructions are repeated several times

Using the ***finally*** keyword, you can possible to factor

```
public class Test {
    public static void main(String[] argv) {
        try {
            ... // Block in which to detect
                    ErrConst and ErrDepl exceptions
        } catch (ErrConst e) {
            System.out.println("Construction error");
        } catch (ErrDepl e) {
            System.out.println("Displacement error");
        } finally {
            System.out.println("End of program");
            System.exit(-1);
        }

    }
}
```

Exception: for or against

- Example: handling errors without exceptions

```
errorType readFile() {
     int errorcode = 0;
     // Open file
     if (isFileIsOpen()) {
          // Determines file length
          if (getFileSize()) {
               // Check memory allocation
               if (getEnoughMemory()) {
                    // Read file from memory
                    if (readFailed())
                        { errorcode = -
                        1;
                    }
               } else {
                    errorcode = -2;
               }
          } else {
               errorcode = -3;
          }

          // Close file
          if (closeTheFileFailed())
               { errorcode = - 4;
          }
     } else {
          errorcode = - 5;
     }
}
```

Error handling becomes very difficult

Difficult to manage function returns

The code is becoming more and more consistent

Exception: for or against

- The exception mechanism enables
  - Conciseness
  - Legibility

```
void readFile() {
    try {
        // Open file
        // Determines file length
        // Check memory allocation
        // Read file from memory
        // Close file
    } catch (FileOpenFailed) {
        ...
    } catch (FileSizeFailed) {
        ...
    } catch (MemoryAllocFailed) {
        ...
    } catch (FileReadFailed) {
        ...
    } catch (FileCloseFailed) {
        ...
    }
}
```

Prefer this solution to the previous one. Clean, professional programming

Exception: common exceptions

- Java provides a number of predefined classes derived from the *Exception* class
- These standard exceptions fall into two categories
  - Explicit exceptions (the ones we've studied), indicated by the ***throws*** keyword
  - Implicit exceptions that are not mentioned by the keyword ***throws***
- List of some exceptions
  - *ArithmeticException* (division by zero)
  - *NullPointerException* (unconstructed reference)
  - *ClassCastException* (cast problem)
  - *IndexOutOfBoundsException* (index overflow problem in table)