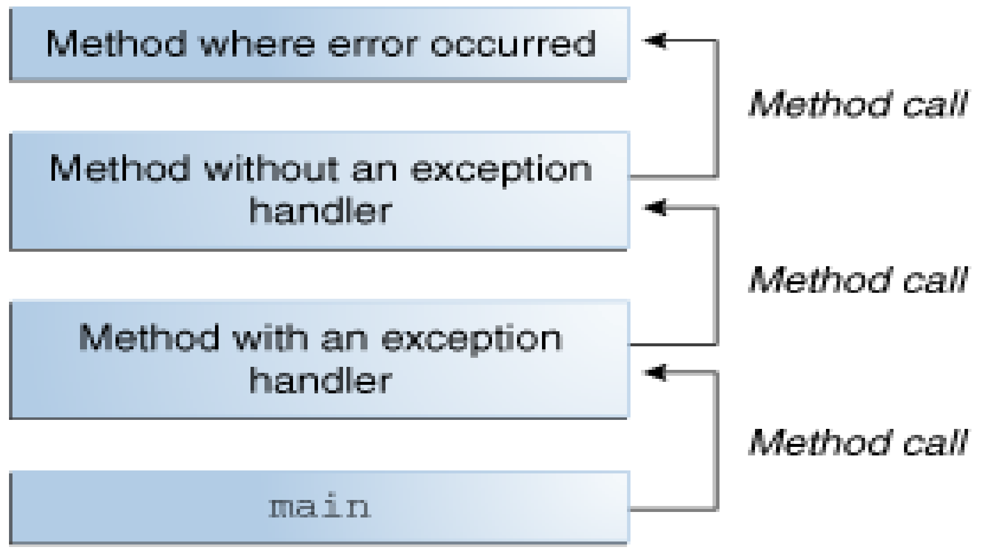
Exceptions

What Is an Exception?

- **Definition:** An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- When an error occurs within a method, the method creates an object and hands it off to the runtime system.
- The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called *throwing an exception*.

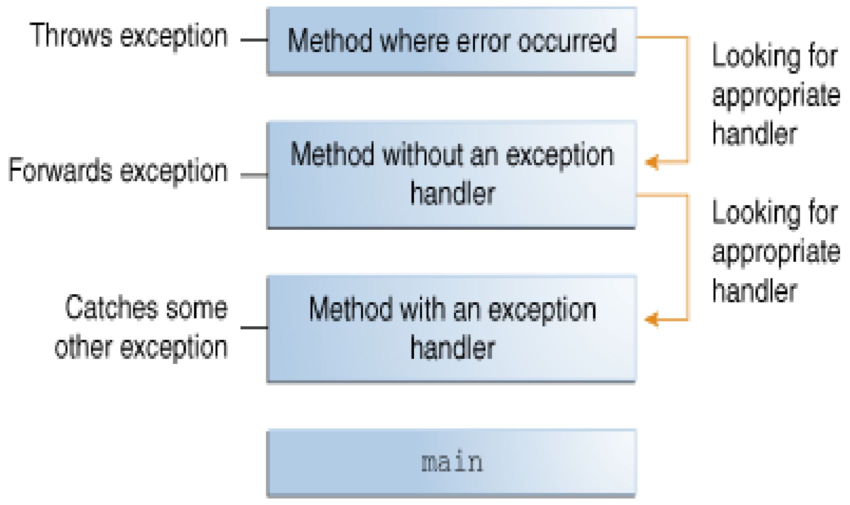
Call Stack


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

Searching the Call Stack for the Exception Handler

- The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*.
- The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called.
- When an appropriate handler is found, the runtime system passes the exception to the handler.
- An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

Call Stack

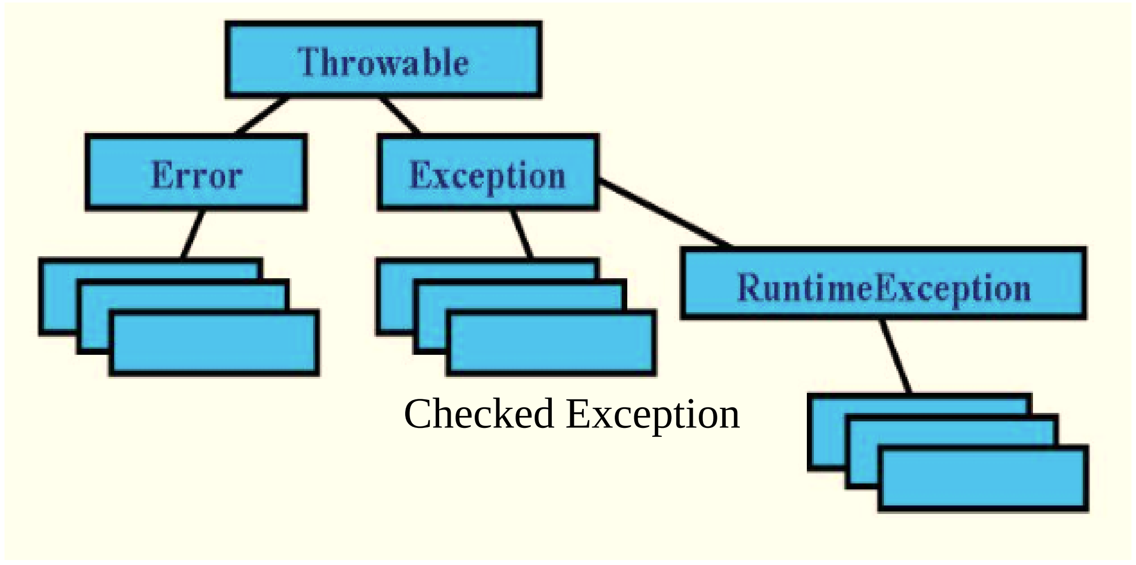

Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

- The exception handler chosen is said to catch the exception.
- If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler the runtime system (and, consequently, the program) terminates.

The Catch or Specify Requirement

- Valid Java programming language code must honor the *Catch or Specify Requirement*.
- The code that might throw certain exceptions must be enclosed by either of the following:
    - A try statement that catches the exception. The try must provide a handler for the exception.
    - A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception.
- Code that fails to honor the Catch or Specify Requirement will not compile.
- Not all exceptions are subject to the Catch or Specify Requirement.

The Three Kinds of Exceptions



Checked Exception

- These are exceptional conditions that a well-written application should anticipate and recover from.
  - For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for java.io.FileReader.
  - Sometimes the user supplies the name of a nonexistent file, and the constructor throws java.io.FileNotFoundException.
  - A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.
- Checked exceptions are subject to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by Error, RuntimeException, and their subclasses.

Error

- These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from.
  - For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction.
  - The unsuccessful read will throw java.io.IOError.
  - An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit.
- Errors are not subject to the Catch or Specify Requirement. Errors are those exceptions indicated by Error and its subclasses

Runtime Exception

- These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from.
- These usually indicate programming bugs, such as logic errors or improper use of an API.
  - For example, consider the application described previously that passes a file name to the constructor for FileReader.
  - If a logic error causes a null to be passed to the constructor, the constructor will throw NullPointerException.
  - The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.
- Runtime exceptions are not subject to the Catch or Specify Requirement.
- Runtime exceptions are those indicated by RuntimeException and its subclasses.

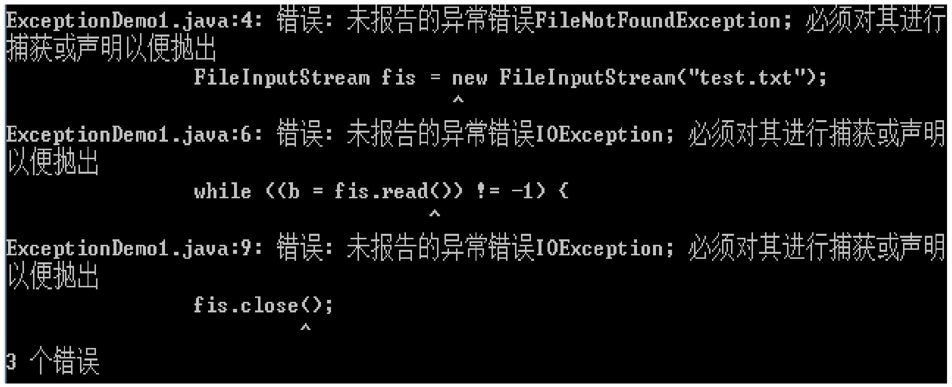
Exceptions

- Errors and runtime exceptions are collectively known as unchecked exceptions.
- Bypassing Catch or Specify
  - Some programmers consider the Catch or Specify Requirement a serious flaw in the exception mechanism and bypass it by using unchecked exceptions in place of checked exceptions.
  - In general, this is not recommended.

Example 1

```
import java.io.*;
class ExceptionDemo1 {
        public static void main(String args[]) {
                FileInputStream fis = new FileInputStream("test.txt");
                int b;
                while ((b = fis.read()) != -1) {
                        System.out.print(b);
                }
                fis.close();
        }
}
```

Compile time output:

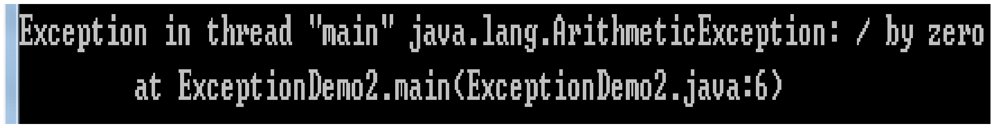


```
ExceptionDemo1.java:4: 错误: 未报告的异常错误FileNotFoundException; 必须对其进行
捕获或声明以便抛出
                FileInputStream fis = new FileInputStream("test.txt");
                                      ^
ExceptionDemo1.java:6: 错误: 未报告的异常错误IOException; 必须对其进行捕获或声明
以便抛出
                while ((b = fis.read()) != -1) {
                                    ^
ExceptionDemo1.java:9: 错误: 未报告的异常错误IOException; 必须对其进行捕获或声明
以便抛出
                fis.close();
                         ^
3 个错误
```

Example 2

```
public class ExceptionDemo2 {
    public static void main(String[] args) {
        int a = 0;
        System.out.println(5/a);
    }
}
```

Runtime Output:



```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionDemo2.main(ExceptionDemo2.java:6)
```

Catching and Handling Exceptions

```
// Note: This class won't compile by design!
import java.io.*;     import java.util.List;   import java.util.ArrayList;
public class ListOfNumbers {
    private List<Integer> list;       private static final int SIZE = 10;
    public ListOfNumbers () {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }
    public void writeList() {
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + list.get(i));
        }
        out.close();
    }
}
```

Catching and Handling Exceptions

- The first line in boldface initializes an output stream on a file.If the file cannot be opened, the constructor throws an IOException.
- The second boldface line is a call to the ArrayList class's get method, which may throws an IndexOutOfBoundsException.
- If you try to compile the ListOfNumbers class, the compiler prints an error message about the exception thrown by the FileWriter constructor.
- However, it does not display an error message about the exception thrown by get.
- The reason is that the exception thrown by the constructor, IOException, is a checked exception, and the one thrown by the get method, IndexOutOfBoundsException, is an unchecked exception.

The try Block

```
try {
   // code
} catch (ExceptionType name) {

} catch (ExceptionType name) {

}finally{

}
```

The try Block

- The segment in the example labeled code contains one or more legal lines of code that could throw an exception.
- To construct an exception handler for the writeList method, enclose the exception-throwing statements of the writeList method within a try block.
  - You can put each line of code that might throw an exception within its own try block and provide separate exception handlers for each.
  - you can put all the writeList code within a single try block and associate multiple handlers with it.
- If an exception occurs within the try block, that exception is handled by an exception handler associated with it. To associate an exception handler with a try block, you must put a catch block after it.

The try and catch Blocks

```
private List<Integer> list;
private static final int SIZE = 10;
PrintWriter out = null;
try {
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + list.get(i));
  }
} catch (IndexOutOfBoundsException  e) {
    System.err.println(" IndexOutOfBoundsException : " + e.getMessage());
    // throw new SampleException(e);
} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

The catch Blocks

- Each catch block is an exception handler and handles the type of exception indicated by its argument.
- The argument type, ExceptionType, declares the type of exception that the handler can handle and must be the name of a class that inherits from the Throwable class.
- The handler can refer to the exception with name.
- The catch block contains code that is executed if and when the exception handler is invoked.

The catch Blocks

- The runtime system invokes the exception handler when the handler is the first one in the call stack whose Exception Type matches the type of the exception thrown.
- The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.
- Exception handlers can do more than just print error messages or halt the program
- They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler using chained exceptions.

The finally Block

- The finally block always executes when the try block exits.
- This ensures that the finally block is executed even if an unexpected exception occurs.
- But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break.
- Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

The finally Block

- The try block of the writeList method that you've been working with here opens a PrintWriter. The program should close that stream before exiting the writeList method.
- The runtime system always executes the statements within the finally block regardless of what happens within the try block. So it's the perfect place to perform cleanup.

```
finally {
    if (out != null) {
    System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

The finally Block

- The finally block is a key tool for preventing resource leaks.
- When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is always recovered.

Putting It All Together

```
public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entering" + " try statement");
        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++)
            out.println("Value at: " + i + " = " + list.get(i));
    } catch (IndexOutOfBoundsException e) {
        System.err.println("Caught IndexOutOfBoundsException: " + e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        } else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

Scenario 1: An Exception Occurs

- When FileWriter throws an IOException, the runtime system immediately stops executing the try block; method calls being executed are not completed.
- The runtime system then starts searching at the top of the method call stack for an appropriate exception handler.
- In this example, when the IOException occurs, the FileWriter constructor is at the top of the call stack.
- However, the FileWriter constructor doesn't have an appropriate exception handler, so the runtime system checks the next method — the writeList method — in the method call stack.

Scenario 1: An Exception Occurs

- The writeList method has two exception handlers: one for IOException and one for IndexOutOfBoundsException.
- The runtime system checks writeList's handlers in the order in which they appear after the try statement.
- The argument to the first exception handler is IndexOutOfBoundsException. This does not match the type of exception thrown.
- The runtime system checks the next exception handler —IOException. This matches the type of exception that was thrown, so the runtime system ends its search for an appropriate exception handler.

Scenario 1: An Exception Occurs

- Now that the runtime has found an appropriate handler, the code in that catch block is executed.
- After the exception handler executes, the runtime system passes control to the finally block.
- Code in the finally block executes regardless of the exception caught above it.
- In this scenario, the FileWriter was never opened and doesn't need to be closed.
- After the finally block finishes executing, the program continues with the first statement after the finallyblock.

Scenario 2: The try Block Exits Normally

- All the statements within the scope of the try block execute successfully and throw no exceptions.
- Execution falls off the end of the try block, and the runtime system passes control to the finally block.
- Because everything was successful, the PrintWriter is open when control reaches the finally block, which closes the PrintWriter.
- Again, after the finally block finishes executing, the program continues with the first statement after the finally block.
- Here is the output from the ListOfNumbers program when no exceptions are thrown.

```
Entering try statement Closing PrintWriter
```

Specifying the Exceptions Thrown by a Method

- Sometimes, it's appropriate for code to catch exceptions that can occur within it.
- In other cases, however, it's better to let a method further up the call stack handle the exception.
- For example, if you were providing the ListOfNumbers class as part of a package of classes, you probably couldn't anticipate the needs of all the users of your package.
- In this case, it's better to not catch the exception and to allow a method further up the call stack to handle it.

Specifying the Exceptions Thrown by a Method

- If the writeList method doesn't catch the checked exceptions that can occur within it, the writeList method must specify that it can throw these exceptions.
- Let's modify the original writeList method to specify the exceptions it can throw instead of catching them.

Specifying the Exceptions Thrown by a Method

- To specify that writeList can throw two exceptions, add a throws clause to the method declaration for the writeList method.
- The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method.
- The clause goes after the method name and argument list and before the brace that defines the scope of the method.

```
public void writeList() throws IOException, IndexOutOfBoundsException {
```

- Remember that IndexOutOfBoundsException is an unchecked exception; including it in the throws clause is not mandatory. You could just write the following.

```
public void writeList() throws IOException {
```

How to Throw Exceptions

- Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception
- Regardless of what throws the exception, it's always thrown with the throw statement.
- The Java platform provides numerous exception classes. All the classes are descendants of the Throwable class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program.
- You can also create your own exception classes to represent problems that can occur within the classes you write.

The throw Statement

- All methods use the throw statement to throw an exception.
- The throw statement requires a single argument: a throwable object.
- Throwable objects are instances of any subclass of the Throwable class.

```
public Object pop() {
    Object obj;
    if (size == 0) {
        throw new EmptyStackException();
    }
    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

The throw Statement

- You can throw only objects that inherit from the java.lang.Throwable class.
- Note that the declaration of the pop method does not contain a throws clause. EmptyStackException is not a checked exception, so pop is not required to state that it might occur.

Throwable Class and Its Subclasses

- **Error Class**

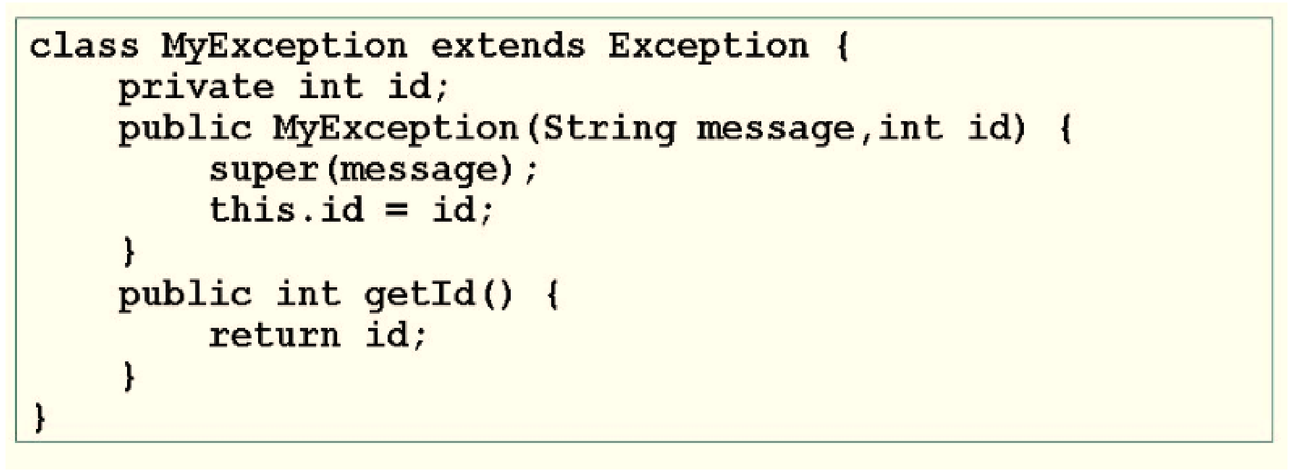
  When a dynamic linking failure or other hard failure in the Java virtual machine occurs, the virtual machine throws an Error. Simple programs typically do *not* catch or throw Errors.

- **Exception Class**
  - Most programs throw and catch objects that derive from the Exception class. An Exception indicates that a problem occurred, but it is not a serious system problem.
  - The Java platform defines the many descendants of the Exception class. These descendants indicate various types of exceptions that can occur.
    - IllegalAccessException signals that a particular method could not be found
    - NegativeArraySizeException indicates that a program attempted to create an array with a negative size.
    - One Exception subclass, RuntimeException, is reserved for exceptions that indicate incorrect use of an API. example : NullPointerException.

Creating Exception Classes

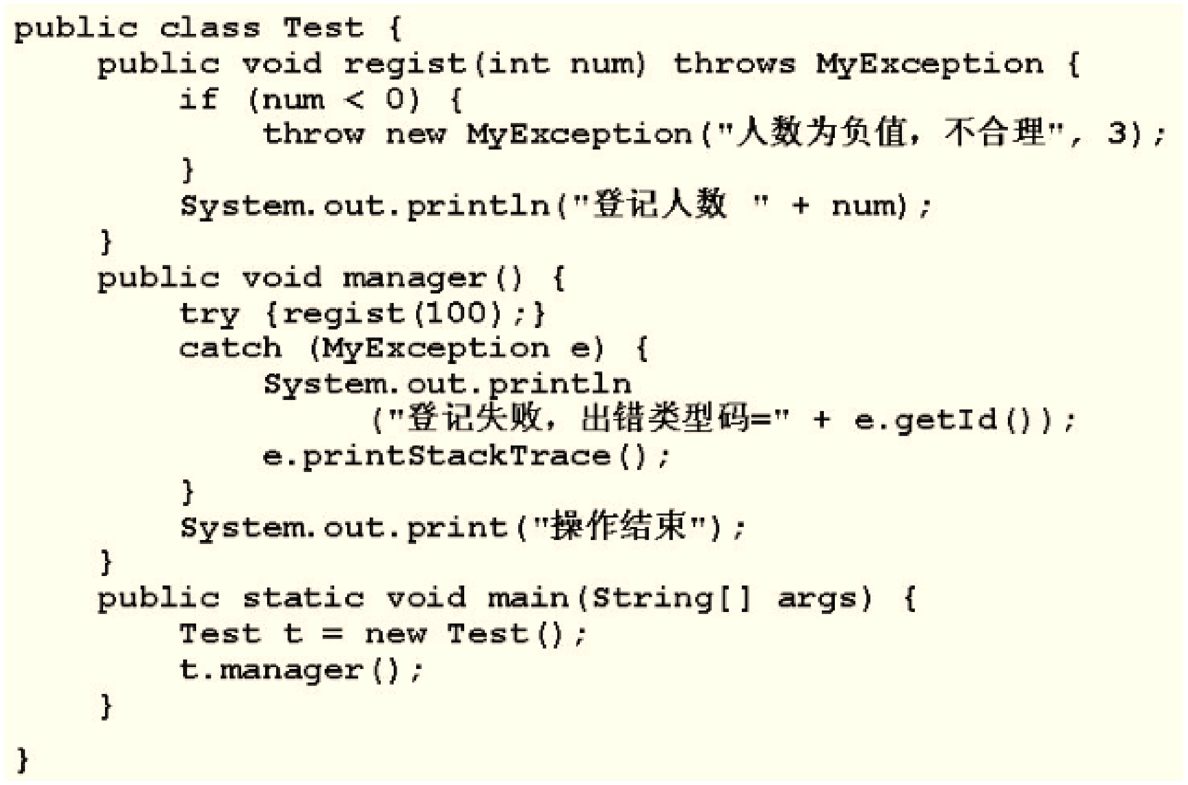
- When faced with choosing the type of exception to throw, you can either use one written by someone else — the Java platform provides a lot of exception classes you can use — or you can write one of your own.
- You should write your own exception classes if you answer yes to any of the following questions; otherwise, you can probably use someone else's.
  - Do you need an exception type that isn't represented by those in the Java platform?
  - Would it help users if they could differentiate your exceptions from those thrown by classes written by other vendors?
  - Does your code throw more than one related exception?
  - If you use someone else's exceptions, will users have access to those exceptions? A similar question is, should your package be independent and self-contained?

Creating Exception Classes



```
class MyException extends Exception {
    private int id;
    public MyException(String message,int id) {
        super(message);
        this.id = id;
    }
    public int getId() {
        return id;
    }
}
```

Creating Exception Classes



```
public class Test {
    public void regist(int num) throws MyException {
        if (num < 0) {
            throw new MyException("人数为负值，不合理", 3);
        }
        System.out.println("登记人数 " + num);
    }
    public void manager() {
        try {regist(100);}
        catch (MyException e) {
            System.out.println
                 ("登记失败，出错类型码=" + e.getId());
            e.printStackTrace();
        }
        System.out.print("操作结束");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.manager();
    }
}
```

Unchecked Exceptions — The Controversy

- If a client can reasonably be expected to recover from an exception, make it a checked exception.
- If a client cannot do anything to recover from the exception, make it an unchecked exception.

Advantages of Exceptions

- Advantage 1: Separating Error-Handling Code from "Regular" Code

```
readFile {
        try {
            open the file;
            determine its size;
            allocate that much memory;
            read the file into memory;
            close the file; }
      catch (fileOpenFailed) {
            doSomething; }
      catch (sizeDeterminationFailed) {
            doSomething; }
      catch (memoryAllocationFailed) {
            doSomething;
      } catch (readFailed) {
            doSomething;
      } catch (fileCloseFailed) {
           doSomething;
      }
  }
```

Advantages of Exceptions

- Advantage 2: Propagating Errors Up the Call Stack

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}
method2 throws exception {
    call method3;
}
method3 throws exception {
    call readFile;
}
```

Advantages of Exceptions

- Advantage 3: Grouping and Differentiating Error Types
  - you can create groups of exceptions and handle exceptions in a general fashion,
  - or you can use the specific exception type to differentiate exceptions and handle exceptions in an exact fashion.

Summary

- A program can use exceptions to indicate that an error occurred.
- To throw an exception, use the throw statement and provide it with an exception object — a descendant of Throwable — to provide information about the specific error that occurred.
- A method that throws an uncaught, checked exception must include a throws clause in its declaration.

Summary

- A program can catch exceptions by using a combination of the try, catch, and finally blocks.
  - The try block identifies a block of code in which an exception can occur.
  - The catch block identifies a block of code, known as an exception handler, that can handle a particular type of exception.
  - The finally block identifies a block of code that is guaranteed to execute, and is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the try block.
  - The try statement should contain at least one catch block or a finally block and may have multiple catch blocks.