

```
oooo
ooooo
ooooooo
oooooooooooooooooooo
oo
```

CS1092: Object-Oriented Programming with Java

More on EXCEPTIONS

Howard Barringer

Room KB2.20/22: email: howard.barringer@manchester.ac.uk

February 2005



Supporting and Background Material

- Copies of key slides (already handed out)
- Chapter 17 of the JTL book (follows this presentation) — [lecture notes](#)
- Recommended reading books for course
 - Chapter 8 (8.2) of Savitch “JAVA: An introduction to Computer Science and Programming”, 3rd Edition
 - Chapter 13 (13.2, 13.5, 13.7) of Liang “Introduction to JAVA Programming”, International Edition



Supporting and Background Material

- Copies of key slides (already handed out)
- Chapter 17 of the JTL book (follows this presentation) — [lecture notes](#)
- Recommended reading books for course
 - Chapter 8 (8.2) of Savitch “JAVA: An introduction to Computer Science and Programming”, 3rd Edition
 - Chapter 13 (13.2, 13.5, 13.7) of Liang “Introduction to JAVA Programming”, International Edition

```
oooo
ooooo
ooooooo
oooooooooooooooooooo
oo
```

Supporting and Background Material

- Copies of key slides (already handed out)
- Chapter 17 of the JTL book (follows this presentation) — [lecture notes](#)
- Recommended reading books for course
 - Chapter 8 (8.2) of Savitch “JAVA: An introduction to Computer Science and Programming”, 3rd Edition
 - Chapter 13 (13.2, 13.5, 13.7) of Liang “Introduction to JAVA Programming”, International Edition

oooo
ooooo
oooooo
oooooooo
oooooooooooooooooooo
oo

Outline

Exception Classes

- Review

- Inheritance in Action

- Custom Exceptions

- The Notional Lottery with Exceptions

- Summary

Outline

Exception Classes

Review

Inheritance in Action

Custom Exceptions

The Notional Lottery with Exceptions

Summary



Brief Recap

- Run-time errors are instances of the **Exception** class



Brief Recap

- Run-time errors are instances of the `Exception` class
- Exceptions can be captured by `try ... catch ...` statement blocks



Brief Recap

- Run-time errors are instances of the `Exception` class
- Exceptions can be captured by `try ... catch ...` statement blocks
- There can be multiple `catch` blocks



Brief Recap

- Run-time errors are instances of the `Exception` class
- Exceptions can be captured by `try ... catch ...` statement blocks
- There can be multiple `catch` blocks
- Exceptions can be explicitly thrown by the `throw` statement



Brief Recap

- Run-time errors are instances of the `Exception` class
- Exceptions can be captured by `try ... catch ...` statement blocks
- There can be multiple `catch` blocks
- Exceptions can be explicitly thrown by the `throw` statement
- Exceptions in GUIs: an unhandled exception in a GUI event thread doesn't terminate the main application



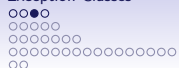
Brief Recap

- Run-time errors are instances of the `Exception` class
- Exceptions can be captured by `try ... catch ...` statement blocks
- There can be multiple `catch` blocks
- Exceptions can be explicitly thrown by the `throw` statement
- Exceptions in GUIs: an unhandled exception in a GUI event thread doesn't terminate the main application
- Methods declare exceptions that are not handled within the method



Brief Recap

- Run-time errors are instances of the **Exception** class
- Exceptions can be captured by **try ... catch ...** statement blocks
- There can be multiple **catch** blocks
- Exceptions can be explicitly thrown by the **throw** statement
- Exceptions in GUIs: an unhandled exception in a GUI event thread doesn't terminate the main application
- Methods declare exceptions that are not handled within the method
- The invoking method of a method that declares an exception is thrown must either declare the exception or handle it



Many sorts of Exceptions

Name some exceptions you've seen raised in your programs ...



Many sorts of Exceptions

Name some exceptions you've seen raised in your programs ...

- `IndexOutOfBoundsException`



Many sorts of Exceptions

Name some exceptions you've seen raised in your programs ...

- `IndexOutOfBoundsException`
- `NumberFormatException`



Many sorts of Exceptions

Name some exceptions you've seen raised in your programs ...

- `IndexOutOfBoundsException`
- `NumberFormatException`
- `NullPointerException`



Many sorts of Exceptions

Name some exceptions you've seen raised in your programs ...

- `IndexOutOfBoundsException`
- `NumberFormatException`
- `NullPointerException`
- `ArithmeticException`



Many sorts of Exceptions

Name some exceptions you've seen raised in your programs ...

- `IndexOutOfBoundsException`
- `NumberFormatException`
- `NullPointerException`
- `ArithmeticException`
- ...

The RuntimeException Class

- Apart from `IOException`, the exceptions we've shown before are subclasses of the `RuntimeException` class
- Which itself is a subclass of the `Exception` class



The RuntimeException Class

- Apart from `IOException`, the exceptions we've shown before are subclasses of the `RuntimeException` class
- Which itself is a subclass of the `Exception` class
- What's special about the `RuntimeException` class?



The RuntimeException Class

- Apart from IOException, the exceptions we've shown before are subclasses of the RuntimeException class
- Which itself is a subclass of the Exception class
- What's special about the RuntimeException class?

Programs do NOT need to catch such exceptions

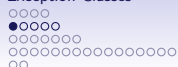


The RuntimeException Class

- Apart from IOException, the exceptions we've shown before are subclasses of the RuntimeException class
- Which itself is a subclass of the Exception class
- What's special about the RuntimeException class?

Programs do NOT need to catch such exceptions

What about IOException? What happens there?



Outline

Exception Classes

Review

Inheritance in Action

Custom Exceptions

The Notional Lottery with Exceptions

Summary



Throwable objects

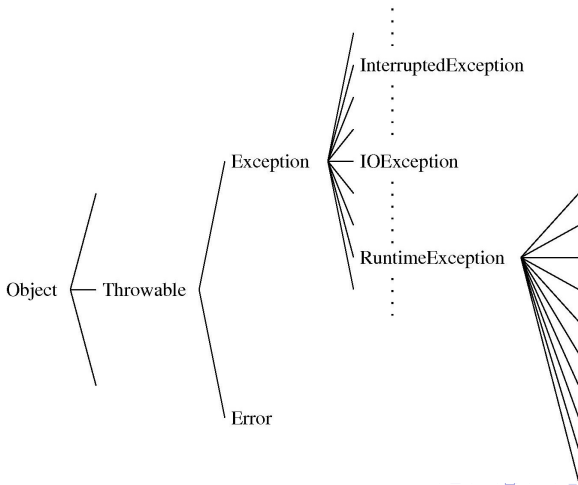
The **Exception** class is a subclass of the **Throwable** class, itself a subclass of **Object**

The **Exception** class has a very large number of subclasses.



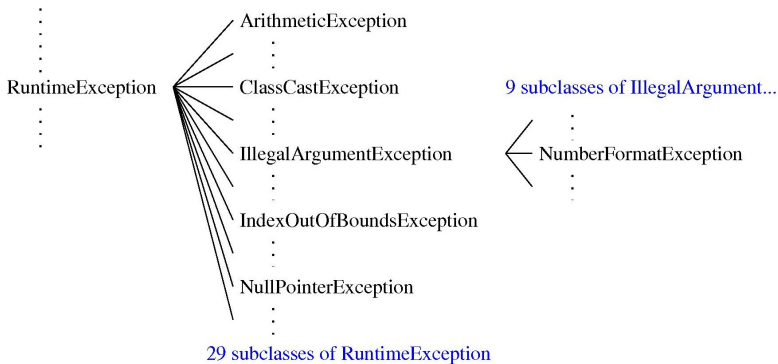
Some of the Exception Class Hierarchy

55 subclasses of Exception!





More of the Exception Class Hierarchy





The **Error** class

The **Error** class is also a subclass of the **Throwable** class.



The **Error** class

The **Error** class is also a subclass of the **Throwable** class.

Errors can also be caught via a **try ... catch ...** block.



The **Error** class

The **Error** class is also a subclass of the **Throwable** class.

Errors can also be caught via a **try ... catch ...** block.

What's the difference between an **Error** and an **Exception**?



The **Error** class

The **Error** class is also a subclass of the **Throwable** class.

Errors can also be caught via a **try ... catch ...** block.

What's the difference between an **Error** and an **Exception**?

Errors, e.g. **NoSuchMethodError** or **OutOfMemoryError**, are typically beyond rectification by the program



The `Error` class

The `Error` class is also a subclass of the `Throwable` class.

Errors can also be caught via a `try ... catch ...` block.

What's the difference between an `Error` and an `Exception`?

Errors, e.g. `NoSuchMethodError` or `OutOfMemoryError`, are typically beyond rectification by the program

Hence, instances of the `Error` class do not need to be caught — rather like `RuntimeException`



The `Error` class

The `Error` class is also a subclass of the `Throwable` class.

Errors can also be caught via a `try ... catch ...` block.

What's the difference between an `Error` and an `Exception`?

Errors, e.g. `NoSuchMethodError` or `OutOfMemoryError`, are typically beyond rectification by the program

Hence, instances of the `Error` class do not need to be caught — rather like `RuntimeException`

However, if an `Error` is caught, it should be re-thrown and allowed to propagate out



Outline

Exception Classes

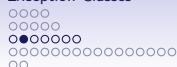
Review

Inheritance in Action

Custom Exceptions

The Notional Lottery with Exceptions

Summary



Defining your own Exceptions

- Java API defines a very large number of exceptions
- BUT the programmer can still define her own
- How and when?
 - typically, for explicit throws of exceptions that don't appropriately relate to any of the API defined exception classes



The `Date` class revisited

Recall the re-usable `Date` class (chapter 14.7)

The constructor was modified to throw an exception for illegal dates

The exception object was an instance of class `Exception`



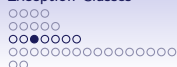
The `Date` class revisited

Recall the re-usable `Date` class (chapter 14.7)

The constructor was modified to throw an exception for illegal dates

The exception object was an instance of class `Exception`

Better practice to create a special `DateException` and then throw an object of that class.



The `Date` class revisited

Recall the re-usable `Date` class (chapter 14.7)

The constructor was modified to throw an exception for illegal dates

The exception object was an instance of class `Exception`

Better practice to create a special `DateException` and then throw an object of that class.

Question: Should we force the exception to be handled?

I.e., should `DateException` be a subclass of:

`Exception` or of `RuntimeException`?



The DateException class

```
public class DateException extends RuntimeException {  
  
    public DateException() {  
        super();  
    }  
  
    public DateException(String message) {  
        super(message);  
    }  
  
}
```



The DateException class

```
public class DateException extends RuntimeException {  
  
    public DateException() {  
        super();  
    }  
  
    public DateException(String message) {  
        super(message);  
    }  
  
}
```

Note: we have not overridden the superclass methods, `getMessage()`, etc.



Other required changes

The **Date** class is then modified by replacing
all occurrences of **Exception**
by **DateException**



Other required changes

The **Date** class is then modified by replacing
all occurrences of **Exception**
by **DateException**

Well, that's not quite right. There's one occurrence that should
become a **RuntimeException**. Can you find it?



The DateDifference Program

```
public class DateDifference {
    public static void main(String [] args) {
        Date date1, date2;
        try {
            date1 = new Date(args[0]);
            date2 = new Date(args[1]);
            System.out.println("From " + date1 + " to " + date2
                               + " is " + date1.daysFrom(date2) + " days");
        } // try
        catch (ArrayIndexOutOfBoundsException exception)
        { System.out.println("Please supply two dates");
          System.err.println(exception); }
        catch (DateException exception)
        { System.out.println(exception.getMessage());
          System.err.println(exception); }
    } // main } // class DateDifference
```



And the difference is ...

Now we obtain:

```
$ java DateDifference 16/12/2004 30/2/2004
```

```
Day 30 must be from 1 to 29 for 2/2004
```

```
DateException: Day 30 must be from 1 to 29 for 2/2004
```

```
$ _
```



And the difference is ...

Now we obtain:

```
$ java DateDifference 16/12/2004 30/2/2004
```

```
Day 30 must be from 1 to 29 for 2/2004
```

```
DateException: Day 30 must be from 1 to 29 for 2/2004
```

```
$ _
```

instead of previously



And the difference is ...

Now we obtain:

```
$ java DateDifference 16/12/2004 30/2/2004
Day 30 must be from 1 to 29 for 2/2004
DateException: Day 30 must be from 1 to 29 for 2/2004
$ _
```

instead of previously

```
$ java DateDifference 16/12/2004 30/2/2004
Day 30 must be from 1 to 29 for 2/2004
java.lang.Exception: Day 30 must be from 1 to 29 for 2/2004
$ _
```

Outline

Exception Classes

Review

Inheritance in Action

Custom Exceptions

The Notional Lottery with Exceptions

Summary



An Exceptional Notional Lottery

Chapter 16 presented the Notional Lottery case study.
It's prime purpose has been to demonstrate inheritance.



An Exceptional Notional Lottery

Chapter 16 presented the Notional Lottery case study.

It's prime purpose has been to demonstrate inheritance.

We can revisit a part, to add exceptions,
and define exceptions as subclasses of other user-defined
exceptions.



An Exceptional Notional Lottery

Chapter 16 presented the Notional Lottery case study.

It's prime purpose has been to demonstrate inheritance.

We can revisit a part, to add exceptions,
and define exceptions as subclasses of other user-defined
exceptions.

We will focus on the abstract class **BallContainer**
and one its subclasses, the **Machine** class.

```
○○○○  
○○○○○  
○○○○○  
○○○○○○○  
○○●○○○○○○○○○○○○○○  
○○
```

About the size of a BallContainer

The constructor for a **BallContainer** is passed the required maximum size.



About the size of a BallContainer

The constructor for a **BallContainer** is passed the required maximum size.

What about a minimum size?

```
○○○○  
○○○○○  
○○○○○○  
○○○○○○○  
○○●○○○○○○○○○○○○○○  
○○
```

About the size of a BallContainer

The constructor for a **BallContainer** is passed the required maximum size.

What about a minimum size?

Sensible to ensure the container can hold at least one ball.



About the size of a BallContainer

The constructor for a **BallContainer** is passed the required maximum size.

What about a minimum size?

Sensible to ensure the container can hold at least one ball.

What should we do if the required maximum size is less than one?



About the size of a BallContainer

The constructor for a **BallContainer** is passed the required maximum size.

What about a minimum size?

Sensible to ensure the container can hold at least one ball.

What should we do if the required maximum size is less than one?

Throw an exception! - A **BallContainerException**

```
○○○○
○○○○○
○○○○○○
○○○○○○○
○○○●○○○○○○○○○○○○○○○○
○○
```

The BallContainer constructor

```
public BallContainer(String requiredName,
                      int requiredMaximumSize)

{

    name = requiredName;
    balls = new Ball[requiredMaximumSize];
    noOfBalls = 0;

}
```


○○○○
○○○○○
○○○○○○○
○○○○○○○
○○○●○○○○○○○○○○○○○
○○

The BallContainer constructor

```
public BallContainer(String requiredName,  
                      int requiredMaximumSize)  
  
{  
  
    if (requiredMaximumSize < 1)  
        throw new BallContainerException  
            ("Size must be at least 1");  
    name = requiredName;  
    balls = new Ball[requiredMaximumSize];  
    noOfBalls = 0;  
  
}
```

○○○○
○○○○○
○○○○○○○
○○○○○○○●○○○○○○○○○○○○○○
○○

The BallContainer constructor

```
public BallContainer(String requiredName,  
                    int requiredMaximumSize)  
    throws BallContainerException  
{  
  
    if (requiredMaximumSize < 1)  
        throw new BallContainerException  
            ("Size must be at least 1");  
    name = requiredName;  
    balls = new Ball[requiredMaximumSize];  
    noOfBalls = 0;  
}
```



The BallContainerException class

```
public class BallContainerException extends RuntimeException
{
    public BallContainerException()
    {
        super();
    }

    public BallContainerException(String message)
    {
        super(message);
    }
}
```

```
○○○○
○○○○○
○○○○○○
○○○○○○○
○○○○○○●○○○○○○○○○○
○○
```

Modification to other methods in the BallContainer class

The methods

`getBall` — there may be no balls in the container

`addBall` — there may be no room in the container

`removeBall` — there may be no ball to remove

`swapBalls` — the specified balls may not exist

all require attention.

```

○○○○
○○○○○
○○○○○○○
○○○○○○○●○○○○○○○○○
○○

```

The original SwapBall method

```

public void swapBalls(int i, int j) throws BallContainerException
{
    if (i >= 0 && i < noOfBalls && j >= 0 && j < noOfBalls)
    {
        Ball oldBallAtI = balls[i];
        balls[i] = balls[j];
        balls[j] = oldBallAtI;
    }
}

```

○○○○
○○○○○
○○○○○○○
○○○○○○○●○○○○○○○○○
○○

The original SwapBall method

```
public void swapBalls(int i, int j) throws BallContainerException
{
    if (i >= 0 && i < noOfBalls && j >= 0 && j < noOfBalls)
    {
        Ball oldBallAtI = balls[i];
        balls[i] = balls[j];
        balls[j] = oldBallAtI;
    }
}
```

○○○○
○○○○○
○○○○○○○
○○○○○○○●○○○○○○○
○○

The new SwapBall method

```
public void swapBalls(int i, int j) throws BallContainerException  
{
```

```
    Ball oldBallAtI = balls[i];  
    balls[i] = balls[j];  
    balls[j] = oldBallAtI;  
}
```



The new SwapBall method

```
public void swapBalls(int i, int j) throws BallContainerException
{
    if (noOfBalls == 0)
        throw new BallContainerException("Cannot swap balls: is empty");

    Ball oldBallAtI = balls[i];
    balls[i] = balls[j];
    balls[j] = oldBallAtI;
}
```




The new SwapBall method

```
public void swapBalls(int i, int j) throws BallContainerException
{
    if (noOfBalls == 0)
        throw new BallContainerException("Cannot swap balls: is empty");

    if (i < 0 || i >= noOfBalls)
        throw new BallContainerException("Swap ball at " + i
            + ": not in range 0.." + (noOfBalls - 1));

    Ball oldBallAtI = balls[i];
    balls[i] = balls[j];
    balls[j] = oldBallAtI;
}
```



The new SwapBall method

```
public void swapBalls(int i, int j) throws BallContainerException
{
    if (noOfBalls == 0)
        throw new BallContainerException("Cannot swap balls: is empty");

    if (i < 0 || i >= noOfBalls)
        throw new BallContainerException("Swap ball at " + i
            + ": not in range 0.." + (noOfBalls - 1));

    if (j < 0 || j >= noOfBalls)
        throw new BallContainerException("Swap ball at " + j
            + ": not in range 0.." + (noOfBalls - 1));

    Ball oldBallAtI = balls[i];
    balls[i] = balls[j];
    balls[j] = oldBallAtI;
}
```

○○○○
○○○○○
○○○○○○
○○○○○○○
○○○○○○○○●○○○○○○○
○○

And now for the Machine

Again, is there a minimum size for a machine.

Sensible for it to hold at least two balls.

○○○○
○○○○○
○○○○○○○
○○○○○○○○●○○○○○
○○

The Machine constructor

```
public Machine(String requiredName,  
                int requiredMaximumSize)  
  
{  
  
    super(requiredName, requiredMaximumSize);  
  
}
```

○○○○
○○○○○
○○○○○○○
○○○○○○○○●○○○○○
○○

The Machine constructor

```
public Machine(String requiredName,  
                int requiredMaximumSize)  
    throws BallContainerException  
{  
  
    super(requiredName, requiredMaximumSize);  
  
}
```

○○○○
○○○○○
○○○○○○○
○○○○○○○○○●○○○○○○○
○○

The Machine constructor

```
public Machine(String requiredName,  
               int requiredMaximumSize)  
    throws BallContainerException  
{  
  
    super(requiredName, requiredMaximumSize);  
  
    if (requiredMaximumSize < 2)  
        throw new MachineException("Size must be at least 2");  
}
```

```
○○○○  
○○○○○  
○○○○○○  
○○○○○○○  
○○○○○○○○○●○○○○○  
○○
```

The MachineException class

MachineException should be a subclass of
BallContainerException



The MachineException class

MachineException should be a subclass of
BallContainerException

```
public class MachineException extends BallContainerException
{
    public MachineException()
    {
        super();
    }

    public MachineException(String message)
    {
        super(message);
    }
}
```


○○○○
○○○○○
○○○○○○○
○○○○○○○○○○●○○○○
○○

The ejectBall method of the Machine class

```
public Ball ejectBall() throws MachineException
{
    try
    {
        int ejectedBallIndex = (int) (Math.random() * getNoOfBalls());

        Ball ejectedBall = getBall(ejectedBallIndex);

        swapBalls(ejectedBallIndex, getNoOfBalls() - 1);
        removeBall();

        return ejectedBall;
    }
    catch (BallContainerException exception){
        throw new MachineException("Cannot eject ball: is empty");
    }
}
```

○○○○
○○○○○
○○○○○○○
○○○○○○○○○○○○●○○○
○○

Testing - I

```
public class TestMachineExceptions
{
    public static void main(String [] args)
    {
        int machineSize = Integer.parseInt(args[0]);
        int fillCount = Integer.parseInt(args[1]);
        int findIndex = Integer.parseInt(args[2]);
        int removeCount1 = Integer.parseInt(args[3]);
        int swapIndex1 = Integer.parseInt(args[4]);
        int swapIndex2 = Integer.parseInt(args[5]);
        int removeCount2 = Integer.parseInt(args[6]);
        int ejectCount = Integer.parseInt(args[7]);
    }
}
```



Testing - II

```
try {  
    System.out.println("Creating machine sized " + machineSize);  
    Machine machine = new Machine("Test4U", machineSize);  
  
    System.out.println("Filling with " + fillCount + " balls");  
    for (int i = 1; i <= fillCount; i++)  
        machine.addBall(new Ball(i, Color.red));  
  
    System.out.println("Finding ball at " + findIndex);  
    machine.getBall(findIndex);  
  
    System.out.println("Adding another ball");  
    machine.addBall(new Ball(fillCount + 1, Color.red));  
  
    System.out.println("Removing " + removeCount1 + " balls");  
    for (int i = 1; i <= removeCount1; i++)  
        machine.removeBall();  
}
```



Testing - III

```
System.out.println("Swapping balls at " + swapIndex1
                  + " and " + swapIndex2);
machine.swapBalls(swapIndex1, swapIndex2);
```

```
System.out.println("Removing " + removeCount2 + " balls");
for (int i = 1; i <= removeCount2; i++)
    machine.removeBall();
```

```
System.out.println("Ejecting " + ejectCount + " balls");
for (int i = 1; i <= ejectCount; i++)
    machine.ejectBall();
```

```
} // try
```

```
catch (Exception exception) {
```

```
    System.out.println("Got exception " + exception);
```

```
} // catch
```

```
} // main
```

```
} // class TestMachineExceptions
```



Testing - Some Results

OK, let's try it out

machine Size	fill Count	find Index	remove Count1	swap Index1	swap Index2	remove Count2	eject Count	expected result
0	-1	-1	-1	-1	-1	-1	-1	Size at least 1
5	5	5	-1	-1	-1	-1	-1	Get Ball at 5: not in range 0..4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮



Outline

Exception Classes

Review

Inheritance in Action

Custom Exceptions

The Notional Lottery with Exceptions

Summary



Summary

- Quick review of exception handling
- Large inheritance structure underneath the `Throwable` class
- Can create custom exception classes
- Rule of thumb:
 1. try to find suitable an existing exception classes
 2. none appropriate, create your own
 3. use inheritance as appropriate
 4. provide two constructors, without and with a message