List of Slides

Java Just in Time

John Latham

February 8, 2018

Chapter 17

Making our own exceptions

Chapter aims

- Standard **exception classes** sometimes not specific enough
  - to model exact nature of exceptions we want.
- Can create our own!
- Look at how **inheritance hierarchy** used to obtain different kinds of exception
  - explore how can have our own.

Section 2

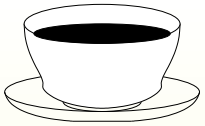
The exception inheritance hierarchy

Aim

AIM: To explain how Java implements the idea of having lots of different kinds of **exception**.

The exception inheritance hierarchy

- Already seen many kinds of **exception**.



Coffee time: How do you think that Java implements the idea of having many kinds of exception?

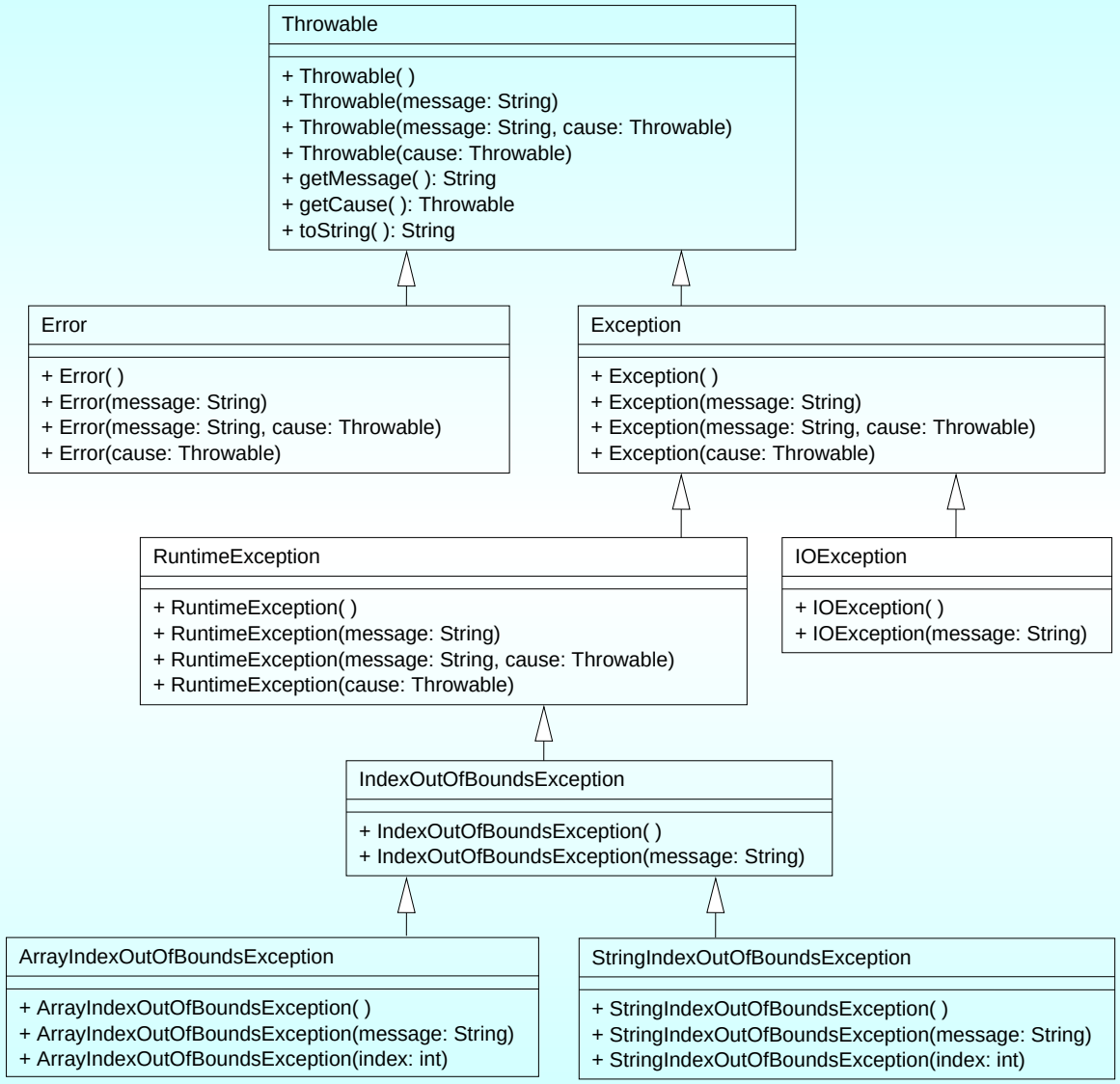
Exception: inheritance hierarchy

- All Java **exception**s modelled as **instance**s of **class**es
  - e.g. `java.lang.Exception` models very general idea of exception
  - `java.lang.ArrayIndexOutOfBoundsException` much more specific kind.
- Different kinds arranged in **inheritance hierarchy**
  - near top models of quite general exceptions
  - near bottom very specific.

Exception: inheritance hierarchy

- Instance of `ArrayIndexOutOfBoundsException` created when **array index** out of legal range.
  - class is **subclass** of `java.lang.IndexOutOfBoundsException`.
- Another subclass of `IndexOutOfBoundsException` is `java.lang.StringIndexOutOfBoundsException`
  - e.g. when supply illegal **method argument** to `charAt()` of a `String`.
- `IndexOutOfBoundsException` is subclass of `java.lang.RuntimeException`
  - kinds of exception we are not *required* to **catch**.
- `java.lang.RuntimeException` is subclass of `Exception`.
- Can show in **UML class diagram**. . . .

Exception: inheritance hierarchy



Exception: inheritance hierarchy

- `Exception` is subclass of `java.lang.Throwable`
  - **type** of all **object**s that can be **thrown** and handled by catches of **try statement**.
- Separate subclass `java.lang.Error`
  - for very serious conditions – usually don't bother trying to catch them
    - e.g. `java.lang.OutOfMemoryError`.

    Can catch these, but are not forced to
    - are **unchecked exception**s.

Exception: inheritance hierarchy

- `Exception` is type of `Throwable`
  - represents conditions that should typically be caught at some point.
- If code in a **method** could cause an `Exception`
  - or one of its subclasses

  compiler forces exception to
  - be caught within method
  - or declared in **throws clause** of the method.

  They are **checked exception**s.

Exception: inheritance hierarchy

- However, `RuntimeException` (and subclasses)
  - represents possible exceptions which programmers usually avoid.
  - E.g. **loop** array index over an array
    - probably get it right, avoid `ArrayIndexOutOfBoundsException`.
- Would be highly inconvenient to *have* to write a **catch clause** or throws clause
  - even when we know the exceptions are avoided.
- Java relaxes the rule for this subclass
  - they too are **unchecked exception**s.
- We must be disciplined, especially in code for **software reuse**
  - *should* write catch or throws clauses if not eliminated possibility.

Exception: inheritance hierarchy

- There are over 70 direct subclasses of `Exception` in the API for Java 7.0
  - including `java.io.IOException`.
- There are almost 50 direct subclasses of `RuntimeException`.

Exception: inheritance hierarchy

- One advantage of **inheritance hierarchy** is when we catch exceptions
  - can decide how general or specific we need to be.
- E.g. following code could cause `ArrayIndexOutOfBoundsException` or `StringIndexOutOfBoundsException`.

```
int arrayIndex, stringIndex;
String[] listOfStrings;

... Code here to populate the above array,
... and set arrayIndex and stringIndex.

char c = listOfStrings[arrayIndex].charAt(stringIndex)
```

Exception: inheritance hierarchy

- Can catch `ArrayIndexOutOfBoundsException`
  - if `arrayIndex` has bad value.
- Can catch `StringIndexOutOfBoundsException`
  - if `stringIndex` has bad value.
- Can have two **catch clause**s, one for each.
  - or could have one catch clause to deal with both
    - catch `IndexOutOfBoundsException`.

The exception inheritance hierarchy

Coffee time: Where does `IllegalArgumentException` fit into the **inheritance hierarchy**? How many **constructor methods** does it have? Find out by looking at the **API** on-line documentation.

Section 3

Example:

The `Date` class with its own exceptions

Aim

AIM: To introduce the idea of making our own **exception**s.

The `Date` class with its own exceptions

- Revisit `Date` from Section **??** starting on page **??**
  - improve by creating and using our own **exception class**.

Exception: making our own exception classes

- Because **exception**s arranged in **inheritance hierarchy**
  - can make our exception **class**es.
- Sometimes classes at bottom of standard exception tree not quite specific for us
  - **design**ed to be appropriate to standard classes.
- Whenever wish to **throw** exception
  - ask ourselves is there standard exception that fits nicely
    - if not – make one.

Exception: making our own exception classes

- How? Choose standard class closest to what we want
  - make **subclass** of it.
- Often will be either `java.lang.Exception`
  - if want ours to be **checked exception**s

  or `java.lang.RuntimeException`.
  - if want ours to be **unchecked exception**s
    - because we believe they can be and typically should be avoided.

Exception: making our own exception classes

- Most often own exception classes contain just four **constructor method**s
  - one with no **method parameter**s
  - one takes `String` for message
  - one takes `String` and **exception cause**
  - one takes just **exception cause**.

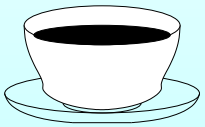
The `Date` class with its own exceptions

- Present `DateException`
- describe changes to `Date`
- develop modified version of `DateDifference`.

The `DateException` class

- `DateException` will **extend** `RuntimeException`
  - **instance**s of `DateException` also instances of `RuntimeException`.
- Thus will be **unchecked exception**s
  - programmers not forced to **catch** them.
- Why?
  - All the erroneous conditions are avoidable, and typically will be avoided.

Coffee time: As an aside, think about programs which read and/or write **file**s. Can errors occur in those scenarios which cannot be avoided by the programmer? Should the **exception**s thus thrown be **checked exception**s or unchecked? (Hint: look at the **UML class diagram**!)



The DateException class

```
001: // Exceptions to be used with the Date class.
002: public class DateException extends RuntimeException
003: {
004:   // Create DateException with no message and no cause.
005:   public DateException()
006:   {
007:     super();
008:   } // DateException
009:
010:
011:   // Create DateException with message but no cause.
012:   public DateException(String message)
013:   {
014:     super(message);
015:   } // DateException
016:
017:
```

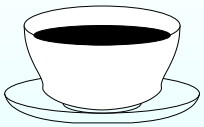
The DateException class

```
018:   // Create DateException with message and cause.
019:   public DateException(String message, Throwable cause)
020:   {
021:     super(message, cause);
022:   } // DateException
023:
024:
025:   // Create DateException with no message but with cause.
026:   public DateException(Throwable cause)
027:   {
028:     super(cause);
029:   } // DateException
030:
031: } // class DateException
```

Coffee time: Why do we have to write these **constructor methods**? Do `DateException` **objects** have any **instance methods**?

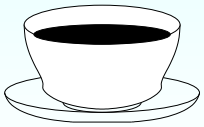
The `Date` class

- Same as version from Section **??** starting on page **??**
  - except most occurrences of `Exception` changed to `DateException`.

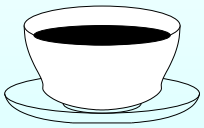


Coffee time: There is one **catch clause** in the `Date` class that must not be changed from `Exception` to `DateException`. Where is it, and why is it not changed like the others?

The Date class



Coffee time: There were some places in the `Date` class from Section **??** where we had to catch an `Exception`, even though we knew that one would never be **throw**n. Is this still necessary with our new version?



Coffee time: Some `Date` instance methods, such as `daysFrom()`, throw a `NullPointerException` when their `Date` **method argument** is the **null reference**. Should these be caught and turned into `DateExceptions`, or left as `NullPointerExceptions`?

The DateDifference class

- Similar to Section **??** on page **??**, but
  - uses `DateException`
  - also structured differently
    - **nested try statement**s,

```
001: // Obtain two dates in day/month/year format from first and second arguments.
002: // Report how many days there are from first to second,
003: // which is negative if first date is the earliest one.
004: public class DateDifference
005: {
006:   public static void main(String[] args)
007:   {
```

The DateDifference class

```
008:    try
009:    {
010:      try
011:      {
012:        if (args.length != 2)
013:          throw new IllegalArgumentException(args.length + " != 2");
014:        Date date1 = new Date(args[0]);
015:        Date date2 = new Date(args[1]);
016:        System.out.println("From " + date1 + " to " + date2 + " is "
017:                           + date1.daysFrom(date2) + " days");
018:      } // try
019:      catch (IllegalArgumentException exception)
020:      {
021:        System.out.println("Please supply exactly two dates");
022:        throw exception;
023:      } // catch
```

The DateDifference class

```
024:      catch (DateException exception)
025:      {
026:        System.out.println("One of your dates has a problem.");
027:        System.out.println(exception.getMessage());
028:        throw exception;
029:      } // catch
030:      catch (Exception exception)
031:      {
032:        System.out.println("Something unforeseen has happened!");
033:        System.out.println(exception.getMessage());
034:        throw exception;
035:      } // catch
036:    } // try
```

The DateDifference class

```
037:    catch (Exception exception)
038:    {
039:      // All exceptions have been already reported to System.out.
040:      System.err.println(exception);
041:      if (exception.getCause() != null)
042:        System.err.println("Caused by: " + exception.getCause());
043:    } // catch
044:  } // main
045:
046: } // class DateDifference
```

Coffee time: Notwithstanding that it sounds like a contradiction in terms, can you foresee what could be an unforeseen **exception**?!

Trying it

Console Input / Output

```
$ java DateDifference 26/03/2017 26/03/2018
From 26/3/2017 to 26/3/2018 is 365 days
$ java DateDifference 26/03/2018 26/03/2017
From 26/3/2018 to 26/3/2017 is -365 days
$ _
```

Run

Console Input / Output

```
$ java DateDifference
Please supply exactly two dates
java.lang.IllegalArgumentException: 0 != 2
$ java DateDifference 26/03/2017
Please supply exactly two dates
java.lang.IllegalArgumentException: 1 != 2
$ java DateDifference 26/03/2017 26/03/2018 ExtraArgument
Please supply exactly two dates
java.lang.IllegalArgumentException: 3 != 2
$ _
```

Run

Trying it

Console Input / Output

```
$ java DateDifference 26/03/2018 "Hello mum"
One of your dates has a problem.
Date ‘Hello mum’ is not in day/month/year format
DateException: Date ‘Hello mum’ is not in day/month/year format
Caused by: java.lang.NumberFormatException: For input string: "Hello mum"
$ java DateDifference 26/03 "Hello mum"
One of your dates has a problem.
Date ‘26/03’ is not in day/month/year format
DateException: Date ‘26/03’ is not in day/month/year format
Caused by: java.lang.ArrayIndexOutOfBoundsException: 2
$ _
```

Run

Trying it

Console Input / Output

```
$ java DateDifference 26/03/2018 03/26/2018
One of your dates has a problem.
Month 26 must be from 1 to 12
DateException: Month 26 must be from 1 to 12
$ java DateDifference 26/03/2018 2018/03/26
One of your dates has a problem.
Year 26 must be >= 1753
DateException: Year 26 must be >= 1753
$ java DateDifference 26/03/2018 30/2/2018
One of your dates has a problem.
Day 30 must be from 1 to 28 for 2/2018
DateException: Day 30 must be from 1 to 28 for 2/2018
$ _
```

Run

A sneaky test?

```
001: // Test DateDifference with a null arguments array!
002: public class DateDifferenceUnforeseenTest
003: {
004:   public static void main(String[] args)
005:   {
006:     DateDifference.main(null);
007:   } // main
008:
009: } // class DateDifferenceUnforeseenTest
```

Console Input / Output

```
$ java DateDifferenceUnforeseenTest
Something unforeseen has happened!
null
java.lang.NullPointerException
$ _
```

Run

Coursework: `GreedyChildren` with exceptions

(Summary only)

Add your own **exception**s to the `GreedyChildren` example.

Section 4

Example: The Notional Lottery with exceptions

Aim

AIM: To reinforce the idea of defining our own **exception**s, and further it by having two of our own exception **class**es, where one is a **subclass** of the other.

The Notional Lottery with exceptions

- Revisit Notional Lottery
  - didn't use exceptions previously.
- E.g. **graphical user interface** accepts input from end user
  - program must check that input is valid.
- Could write checking code in GUI
  - perhaps better to have model classes check validity
    - **throw** exceptions which GUI classes **catch**.
- Look only at `BallContainer` and `Machine`.
  - develop `BallContainerException` and `MachineException`
  - alter `BallContainer` and `Machine` to use them.
- Also present program `TestMachineExceptions`.

The `BallContainerException` class

- `BallContainerException` is subclass of `Exception`
  - **instance**s are **checked exception**s.
- Instance **throw**n when invalid operation performed on `BallContainer`
  - e.g. removing ball when empty.

The BallContainerException class

- Like DateException: but called BallContainerException, extends Exception.

```
001: // Exceptions to be used with the BallContainer class.
002: public class BallContainerException extends Exception
003: {
004:    // Create BallContainerException with no message and no cause.
005:    public BallContainerException()
006:    {
007:      super();
008:    } // BallContainerException
009:
010:
011:    // Create BallContainerException with message but no cause.
012:    public BallContainerException(String message)
013:    {
014:      super(message);
015:    } // BallContainerException
```

The BallContainerException class

```
016:
017:
018:    // Create BallContainerException with message and cause.
019:    public BallContainerException(String message, Throwable cause)
020:    {
021:      super(message, cause);
022:    } // BallContainerException
023:
024:
025:    // Create BallContainerException with no message but with cause.
026:    public BallContainerException(Throwable cause)
027:    {
028:      super(cause);
029:    } // BallContainerException
030:
031: } // class BallContainerException
```

The `MachineException` class

- `MachineException` is **subclass** of `BallContainerException`
  - thrown when invalid operation specific to machine-like behaviour is performed
    - e.g. *eject* ball when machine is empty (`ejectBall()` is in `Machine`, but not `BallContainer`).
- `BallContainerExceptions` thrown by code inside `BallContainer` class
  - `MachineExceptions` by code inside `Machine` class.
- Thus `Machines` can **throw** both kinds of **exception**.

The MachineException class

```
001: // Exceptions to be used with the Machine class.
002: public class MachineException extends BallContainerException
003: {
004:    // Create MachineException with no message and no cause.
005:    public MachineException()
006:    {
007:      super();
008:    } // MachineException
009:
010:
011:    // Create MachineException with message but no cause.
012:    public MachineException(String message)
013:    {
014:      super(message);
015:    } // MachineException
016:
017:
```

The MachineException class

```
018:    // Create MachineException with message and cause.
019:    public MachineException(String message, Throwable cause)
020:    {
021:      super(message, cause);
022:    } // MachineException
023:
024:
025:    // Create MachineException with no message but with cause.
026:    public MachineException(Throwable cause)
027:    {
028:      super(cause);
029:    } // MachineException
030:
031: } // class MachineException
```

The BallContainer class

```
001: // Representation of a container of balls for the lottery,
002: // with a fixed size and zero or more balls in a certain order.
003: public abstract class BallContainer
004: {
005:   // The name of the BallContainer.
006:   private final String name;
007:
008:   // The balls contained in the BallContainer.
009:   private final Ball[] balls;
010:
011:   // The number of balls contained in the BallContainer.
012:   // These are stored in balls, indexes 0 to noOfBalls - 1.
013:   private int noOfBalls;
```

The BallContainer class

- Size must be at least 1.

```
016:   // Constructor is given the name and size.
017:   public BallContainer(String requiredName, int requiredSize)
018:         throws BallContainerException
019:   {
020:     if (requiredSize < 1)
021:       throw new BallContainerException("Size must be at least 1");
022:     name = requiredName;
023:     balls = new Ball[requiredSize];
024:     noOfBalls = 0;
025:   } // BallContainer
```

The BallContainer class

- These bits the same.

```
028:   // Returns the BallContainer's name.
029:   public String getName()
030:   {
031:     return name;
032:   } // getName
033:
034:
035:   // Returns the name of the type of BallContainer.
036:   public abstract String getType();
```

The BallContainer class

- getBall() **throws exceptions**.

```
039:   // Returns the Ball at the given index in the BallContainer.
040:   // Throws exception if that index is not in the range 0 to noOfBalls - 1.
041:   public Ball getBall(int index) throws BallContainerException
042:   {
043:     if (noOfBalls == 0)
044:       throw new BallContainerException("Cannot get ball: is empty");
045:
046:     if (index < 0 || index >= noOfBalls)
047:       throw new BallContainerException
048:                   ("Get ball at " + index + ": not in range 0.."
049:                    + (noOfBalls - 1));
050:     return balls[index];
051:   } // getBall;
```

The BallContainer class

- These bits the same.

```
054:   // Returns the number of balls in the BallContainer.
055:   public int getNoOfBalls()
056:   {
057:     return noOfBalls;
058:   } // getNoOfBalls
059:
060:
061:   // Returns the size of the BallContainer.
062:   public int getSize()
063:   {
064:     return balls.length;
065:   } // getSize
```

The BallContainer class

- addBall() **throws exceptions**.

```
068:  // Adds the given ball into the BallContainer, at the next highest unused
069:  // index position. Throws exception if the BallContainer is full.
070:  public void addBall(Ball ball) throws BallContainerException
071:  {
072:    if (noOfBalls == balls.length)
073:      throw new BallContainerException("Cannot add ball: is full");
074:    balls[noOfBalls] = ball;
075:    noOfBalls++;
076:  } // addBall
```

The BallContainer class

- swapBalls() **throws exceptions**.

```
079:   // Swaps the balls at the two given index positions.
080:   // Throws exception if either index is not in the range 0 to noOfBalls - 1.
081:   public void swapBalls(int index1, int index2) throws BallContainerException
082:   {
083:     if (noOfBalls == 0)
084:       throw new BallContainerException("Cannot swap balls: is empty");
085:
086:     if (index1 < 0 || index1 >= noOfBalls)
087:       throw new BallContainerException
088:                   ("Swap ball at " + index1 + ": not in range 0.."
089:                    + (noOfBalls - 1));
090:
```

The BallContainer class

```
091:    if (index2 < 0 || index2 >= noOfBalls)
092:      throw new BallContainerException
093:                    ("Swap ball at " + index2 + ": not in range 0.."
094:                     + (noOfBalls - 1));
095:
096:    Ball thatWasAtIndex1 = balls[index1];
097:    balls[index1] = balls[index2];
098:    balls[index2] = thatWasAtIndex1;
099:  } // swapBalls;
```

The BallContainer class

- removeBall() **throws exceptions**.

```
102:   // Removes the Ball at the highest used index position.
103:   // Throws exception if the BallContainer is empty.
104:   public void removeBall() throws BallContainerException
105:   {
106:     if (noOfBalls <= 0)
107:       throw new BallContainerException("Cannot remove ball: is empty");
108:     noOfBalls--;
109:   } // removeBall
```

The BallContainer class

- toString() same as before
  - except use **override annotation**.

```
112:   // Mainly for testing.
113:   @Override
114:   public String toString()
115:   {
116:     String result = getType() + " " + name + "(<=" + balls.length + ")";
117:     for (int index = 0; index < noOfBalls; index++)
118:       result += String.format("%n%d %s", index, balls[index]);
119:     return result;
120:   } // toString
121:
122: } // class BallContainer
```

The Machine class

- Validity checks specific to machines
  - e.g. minimum size is two!

```
001: // Representation of a lottery machine,
002: // with the facility for a randomly chosen ball to be ejected.
003: public class Machine extends BallContainer
004: {
005:   // Constructor is given the name and size.
006:   public Machine(String name, int size) throws BallContainerException
007:   {
008:     super(name, size);
009:     if (size < 2)
010:       throw new MachineException("Size must be at least 2");
011:   } // Machine
```

The Machine class

Coffee time: What would be the result of the code `new Machine("Empty", 0)`? What would be the result of `new Machine("Single", 1)`? What if we were to swap the two **statement**s in the constructor method?

Coffee time: Why did we declare that the **constructor method throws** `BallContainerException` rather than `MachineException`? What would happen if we accidentally said it throws `MachineException`? What if `BallContainerException` was a **subclass** of `RuntimeException` rather than `Exception`?

The Machine class

```
014:   // Returns the name of the type of BallContainer.
015:   public String getType()
016:   {
017:     return "Lottery machine";
018:   } // getType
```

- ejectBall() **catch**es BallContainerException
  - and **throw**s MachineException, with BallContainerException as cause. . .

The Machine class

```
021:   // Randomly chooses a ball in the machine, and ejects it.
022:   // The ejected ball is returned. If the machine is empty then
023:   // it throws an exception.
024:   public Ball ejectBall() throws MachineException
025:   {
026:     try
027:     {
028:       // Math.random() * getNoOfBalls yields a number
029:       // which is >= 0 and < number of balls.
030:       int ejectedBallIndex = (int) (Math.random() * getNoOfBalls());
031:
032:       Ball ejectedBall = getBall(ejectedBallIndex);
033:
034:       swapBalls(ejectedBallIndex, getNoOfBalls() - 1);
035:       removeBall();
036:
037:       return ejectedBall;
```

The Machine class

```
038:    } // try
039:    catch (BallContainerException exception)
040:    {
041:      throw new MachineException("Cannot eject ball: is empty", exception);
042:    } // catch
043:  } // ejectBall
044:
045: } // class Machine
```

The TestMachineExceptions class

```
001: import java.awt.Color;
002:
003: /* For testing BallContainer and Machine with BallContainerException and
004:    MachineException. Depending on the values given, it will produce exceptions
005:    at different points, which we catch and print out. By running it with
006:    different values, we are able to test every possible throw statement in
007:    BallContainer and Machine.
008:  */
009: public class TestMachineExceptions
010: {
011:   public static void main(String[] args)
012:   {
```

The TestMachineExceptions class

```
013:     int machineSize = Integer.parseInt(args[0]);
014:     int fillCount = Integer.parseInt(args[1]);
015:     int findIndex = Integer.parseInt(args[2]);
016:     int removeCount1 = Integer.parseInt(args[3]);
017:     int swapIndex1 = Integer.parseInt(args[4]);
018:     int swapIndex2 = Integer.parseInt(args[5]);
019:     int removeCount2 = Integer.parseInt(args[6]);
020:     int ejectCount = Integer.parseInt(args[7]);
021:
022:     try
023:     {
```

The TestMachineExceptions class

```
024:        System.out.println("Creating machine sized " + machineSize);
025:        Machine machine = new Machine("Test4U", machineSize);
026:
027:        System.out.println("Filling with " + fillCount + " balls");
028:        for (int i = 1; i <= fillCount; i++)
029:          machine.addBall(new Ball(i, Color.red));
030:
031:        System.out.println("Finding ball at " + findIndex);
032:        machine.getBall(findIndex);
033:
034:        System.out.println("Adding another ball");
035:        machine.addBall(new Ball(fillCount + 1, Color.red));
036:
```

The TestMachineExceptions class

```
037:        System.out.println("Removing " + removeCount1 + " balls");
038:        for (int i = 1; i <= removeCount1; i++)
039:          machine.removeBall();
040:
041:        System.out.println("Swapping balls at " + swapIndex1
042:                           + " and " + swapIndex2);
043:        machine.swapBalls(swapIndex1, swapIndex2);
044:
045:        System.out.println("Removing " + removeCount2 + " balls");
046:        for (int i = 1; i <= removeCount2; i++)
047:          machine.removeBall();
048:
049:        System.out.println("Ejecting " + ejectCount + " balls");
050:        for (int i = 1; i <= ejectCount; i++)
051:          machine.ejectBall();
```

The TestMachineExceptions class

```
052:
053:     } // try
054:     catch (Exception exception)
055:     {
056:       System.out.println("Got exception " + exception);
057:       if (exception.getCause() != null)
058:         System.out.println("Caused by: " + exception.getCause());
059:     } // catch
060:   } // main
061:
062: } // class TestMachineExceptions
```

Trying it

No	Size	Fill	Find	Rem	Swap	Rem	Eject	Expected result
1	0	-1	-1	-1	-1, -1	-1	-1	BCE: Size must be at least 1
2	1	-1	-1	-1	-1, -1	-1	-1	ME: Size must be at least 2
3	5	0	1	-1	-1, -1	-1	-1	BCE: Cannot get ball: is empty
4	5	5	5	-1	-1, -1	-1	-1	BCE: Get ball at 5: not in range 0..4
5	5	5	4	-1	-1, -1	-1	-1	BCE: Cannot add ball: is full
6	5	1	0	2	0, 0	-1	-1	BCE: Cannot swap balls: is empty
7	5	4	3	0	-1, 0	-1	-1	BCE: Swap ball at -1: not in range 0..4
8	5	4	3	0	0, 5	-1	-1	BCE: Swap ball at 5: not in range 0..4
9	5	3	2	0	0, 1	5	-1	BCE: Cannot remove ball: is empty
10	5	3	2	0	0, 1	0	5	ME: Cannot eject ball: is empty

Trying it

Console Input / Output

```
$ java TestMachineExceptions 0 -1 -1 -1 -1 -1 -1 -1
Creating machine sized 0
Got exception BallContainerException: Size must be at least 1
$ java TestMachineExceptions 1 -1 -1 -1 -1 -1 -1 -1
Creating machine sized 1
Got exception MachineException: Size must be at least 2
$ java TestMachineExceptions 5 0 1 -1 -1 -1 -1 -1
Creating machine sized 5
Filling with 0 balls
Finding ball at 1
Got exception BallContainerException: Cannot get ball: is empty
$ java TestMachineExceptions 5 5 5 -1 -1 -1 -1 -1
Creating machine sized 5
Filling with 5 balls
Finding ball at 5
Got exception BallContainerException: Get ball at 5: not in range 0..4
$ _
```

Run

Trying it

Console Input / Output

```
$ java TestMachineExceptions 5 5 4 -1 -1 -1 -1 -1
Creating machine sized 5
Filling with 5 balls
Finding ball at 4
Adding another ball
Got exception BallContainerException: Cannot add ball: is full
$ java TestMachineExceptions 5 1 0 2 0 0 -1 -1
Creating machine sized 5
Filling with 1 balls
Finding ball at 0
Adding another ball
Removing 2 balls
Swapping balls at 0 and 0
Got exception BallContainerException: Cannot swap balls: is empty
$ _
```

Run

Trying it

Console Input / Output

```
$ java TestMachineExceptions 5 4 3 0 -1 0 -1 -1
Creating machine sized 5
Filling with 4 balls
Finding ball at 3
Adding another ball
Removing 0 balls
Swapping balls at -1 and 0
Got exception BallContainerException: Swap ball at -1: not in range 0..4
$ java TestMachineExceptions 5 4 3 0 0 5 -1 -1
Creating machine sized 5
Filling with 4 balls
Finding ball at 3
Adding another ball
Removing 0 balls
Swapping balls at 0 and 5
Got exception BallContainerException: Swap ball at 5: not in range 0..4
$ _
```

Run

Trying it

Console Input / Output

```
$ java TestMachineExceptions 5 3 2 0 0 1 5 -1
Creating machine sized 5
Filling with 3 balls
Finding ball at 2
Adding another ball
Removing 0 balls
Swapping balls at 0 and 1
Removing 5 balls
Got exception BallContainerException: Cannot remove ball: is empty
$ java TestMachineExceptions 5 3 2 0 0 1 0 5
Creating machine sized 5
Filling with 3 balls
Finding ball at 2
Adding another ball
Removing 0 balls
Swapping balls at 0 and 1
Removing 0 balls
Ejecting 5 balls
Got exception MachineException: Cannot eject ball: is empty
Caused by: BallContainerException: Cannot get ball: is empty
$ _
```

Run

Coursework: `MobileIceCreamParlour` with exceptions

(Summary only)

Add a **subclass** of your own **exception** to the `GreedyChildren` example.

Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
  - a self-test question,
  - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.