# Exceptions & Miscellaneous
# Lecture 17

Waterford Institute of Technology

April 6, 2016

John Fitzgerald

# Presentation outline

Estimated duration presentation

Questions at end presentation

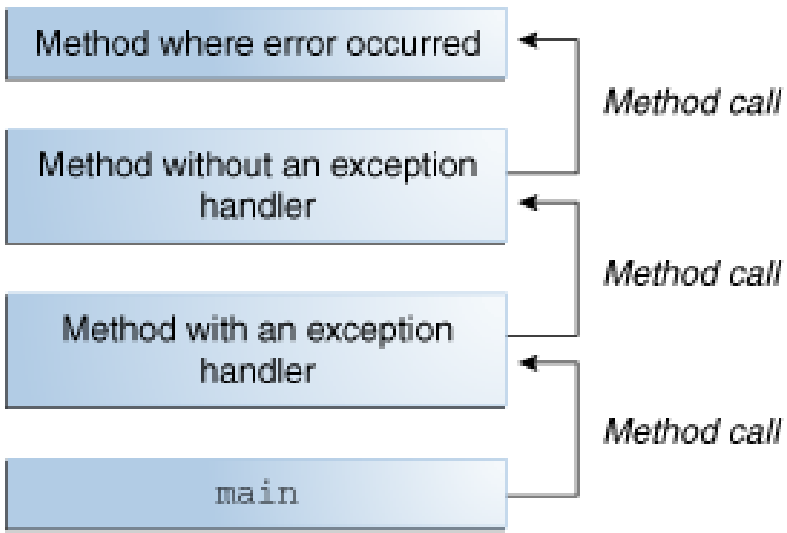
Topics discussed:

- Exceptions
- Brief mention outstanding topics

# Exception

## Disruptive runtime event

- execution halts
- *exception object* created
  - contains error information
  - handed to runtime system
  - called *throwing exception*
  - runtime seeks solution

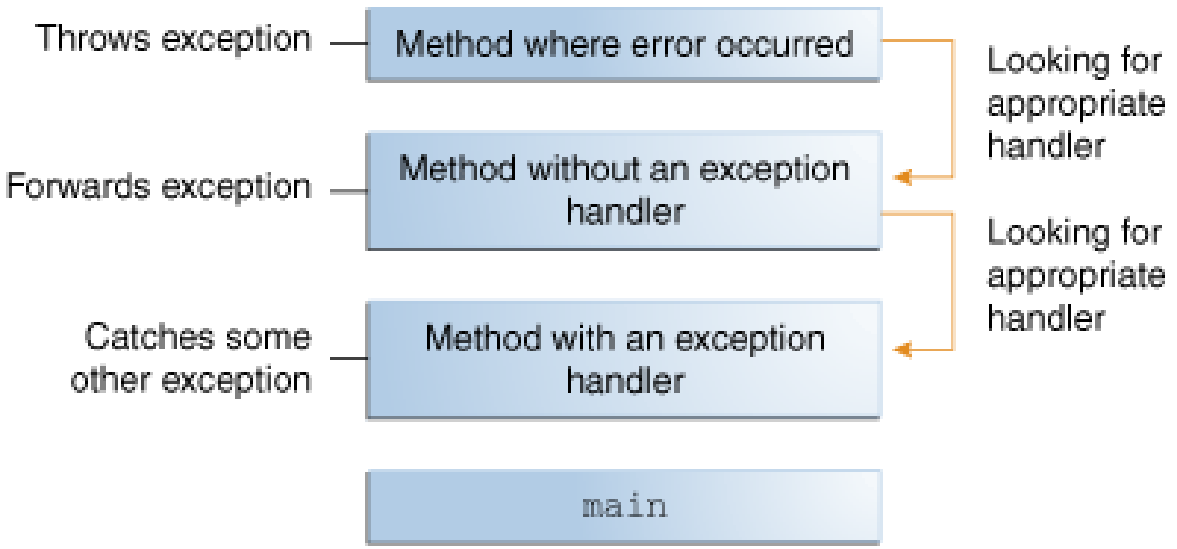



The call stack.

# Exception

## Exception handler: Runtime searches call stack for code to handle exception

- *exception handler*
  - code block
  - suitable location



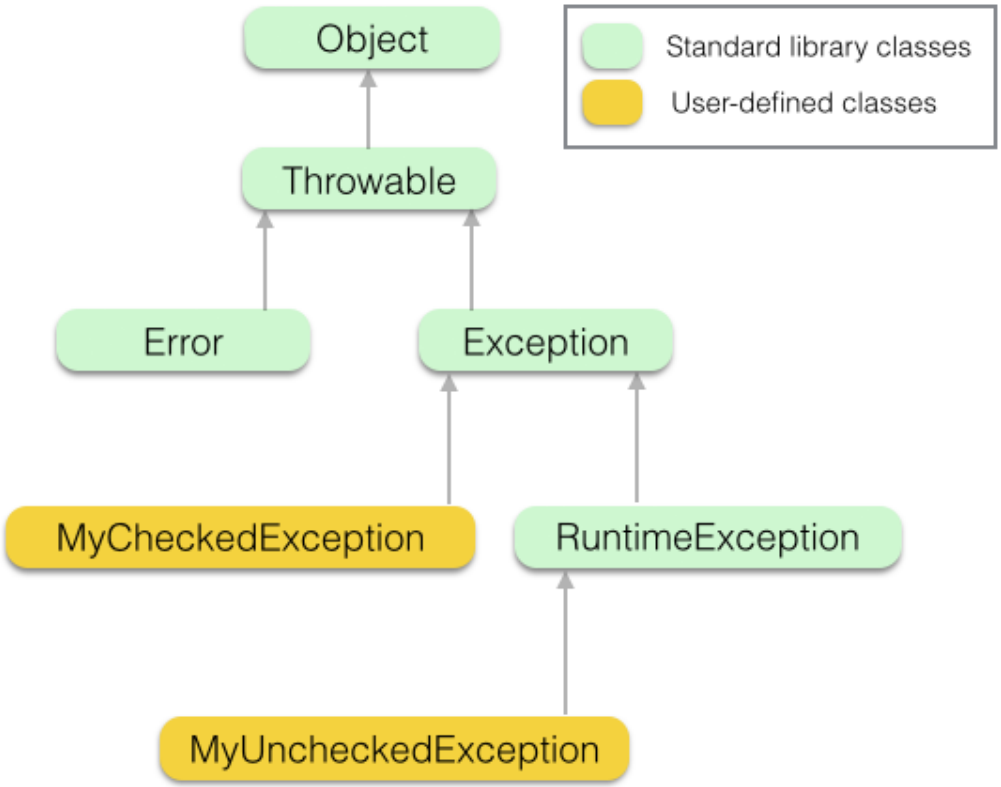

Searching the call stack for the exception handler.

# Exceptions

## Definition

**Exception** a disruptive event during program execution

- Managed in Java with Exception classes
- Derived from Throwable
- Exceptions are:
  - *Checked*
  - *Unchecked*



# Exceptions

## Checked v Unchecked

**Checked exception**

- Subclass of Exception
- Handle where occurs or throw
- Compiler enforces rules

**Unchecked exception**

- RuntimeException subclass
- No compiler-enforced rules
- When, where, if to handle optional

# Exceptions

## Checked example

### Checked exception

- Consider class java.io.FileWriter
- *public FileWriter(String fileName) throws IOException*
- Handle exception within method

```
public static void append(String data, String filename)
{
  try {
    FileWriter fw = new FileWriter(filename,true);
    fw.write(data +"\n");
    fw.close();
  }
  catch (IOException e) { e.printStackTrace(); }
}

public static void main(String[] args) {
    append("data", "data.txt");
}
```

# Exceptions

## Checked example

### Checked exception

- Consider class java.io.FileWriter
- *public FileWriter(String fileName) throws IOException*
- Handle exception in *main* method (better)

```
public static void append(String data, String filename) throws IOException
{
  FileWriter fw = new FileWriter(filename,true);
  fw.write(data +"\n");
  fw.close();
}

public static void main(String[] args) {

  try {
    append("data", "data.txt");
  } catch (IOException e) {
    e.printStackTrace(); }
}
```

# Exceptions

## Unchecked example

### NullPointerException

- Library class runtime exception
- Attempt access through **null** reference
- No *check* necessary

```
Map<String, Book> books;

public void triggerNullPointerException()
{
  books.get(0);
}
```

```
Exception in thread "main" java.lang.NullPointerException
        at exceptions.ExceptionDemo.triggerNullPointerException(ExceptionDemo.java:11)
        at exceptions.ExceptionTest.main(ExceptionTest.java:10)
```

# Exceptions

## Summary unchecked

- Easier to use than checked
- Few rule checks made by compiler
- *RuntimeException* or a subclass
- Exception handler optional

```
public static void main(String[] args)
{
    Map<String, Book> books;
    ...
    Book book = books.getBook("Joyce");
    ...
}
```

```
// Bad coding style – clutter
public Book getBook(String key)
{
    if(key == null)
    {
        throw NullPointerException
    }
    return books.get(key);
}
```

# Exceptions

## Checked example

### BookException

- User-defined class
- Attempt access map using **null** key
- Must provide infrastructure to *check*

```
public Book getBook(String key) throws BookException
{
  if(key == null)
  {
    throw new BookException("\nChecked exception demo: key is null");
  }
  return books.get(key);
}
```

```
exceptions.BookException:

Checked exception demo: key is null
        at exceptions.ExceptionDemo.getDetailsChecked(ExceptionDemo.java:26)
        at exceptions.ExceptionTest.main(ExceptionTest.java:17)
```

# Exceptions

## Check infrastructure

### throws throw

- Signature method in which checked exception thrown

```
public Book getBook(String key) throws BookException
```

- **throw** exception if key null

```
throw new BookException("\nChecked exception demo: key is null");
```

```
public Book getBook(String key) throws BookException
{
  if(key == null)
  {
    throw new BookException("\nChecked exception demo: key is null");
  }
  return books.get(key);
}
```

# Exceptions

## Check infrastructure

### try catch

- Wrap method invocation in *try* block

  try {demo.getBook(null); }

- Wrap action in event of exception in *catch* block

  catch (BookException e) {e.printStackTrace(); }

```
public static void main(String[] args)
{
  try
  {
    demo.getBook(null);
  }
  catch (BookException e)
  {
    e.printStackTrace();
  }
}
```

# Exceptions

## Check infrastructure

### finally

- Where present, executes on exit from *try*
  - Additionally to exception handling,
  - Helps avoid resource leaks because
  - Block executed even when exception thrown

```
Out out = new Out();
// write to stdout
try{
  out.println("Test 1");
}
finally
{
  if(out != null)
  {
    out.close();
  }
}
```

# Exceptions

## User-defined BookException class

Convention is to end name with *Exception*
May optionally override *Throwable* methods such as:

- getLocalizedMessage
- getMessage
- printStackTrace

```
public class BookException extends Exception
{
  String message;
  BookException(String message) {
    this.message = message;
  }
  public String getLocalizedMessage() {
    return "\n" + message;
  }
}
```

# Outstanding topics

## Generics

Generics: a big topic

Our introduction restricted to implementation of

- List
- Map
- Set

```
ArrayList<Message> m;
HashMap<Integer, ArrayList<Bid>> bids;
HashSet<Person> members;
```

# Outstanding topics

## Generics

*Generic type* is generic class parameterized over types.

```
public class Pair<X, Y>
{
  X x;
  Y y;
  public Pair(X x, Y y) {
    this.x = x;
    this.y = y;
  }
}
```

```
Pair<Integer, Integer> pairxy
  = new<Integer, Integer> Pair(10,
      20);
```

```
Person p1 = new Person (40);
Person p2 = new Person (45);
Pair<Person, Person> persons
  = new<Person, Person> Pair(p1, p2);
```

# Outstanding topics

## Generics - wildcard with upper bound

```
public class Main {
  public static void main(String[] args) {
    List<Shape> shapes = new ArrayList<>();
    for (int i = 0; i < 5; i +=1 )
      shapes.add(new Circle(10));
    for (int i = 0; i < 5; i +=1 )
      shapes.add(new Rectangle(10, 20));
    Canvas.drawAll(shapes);
  }
}
```

```
public class Canvas {
  // List element is Shape or subtype of Shape
  static void drawAll(List<? extends Shape> shapes) {
    for (Shape shape : shapes)
      System.out.println("Drawing " + shape.getClass());
  }
}
```

# Outstanding topics

## Generics - wildcard with lower bound

```
// ? super T is some type which is T or super class of T
Collections.sort(List<T> list, Comparator<? super T> c))
```

```
import java.util.Comparator;
public class MessageTextComparator implements Comparator<Message> {
    @Override
    public int compare(Message o1, Message o2) {
        return o1.messageText.compareTo(o2.messageText);
    }
}
```

```
ArrayList<Message> messages = new ArrayList<>();
...
Collections.sort(messages, new MessageTextComparator());
```

# Outstanding topics

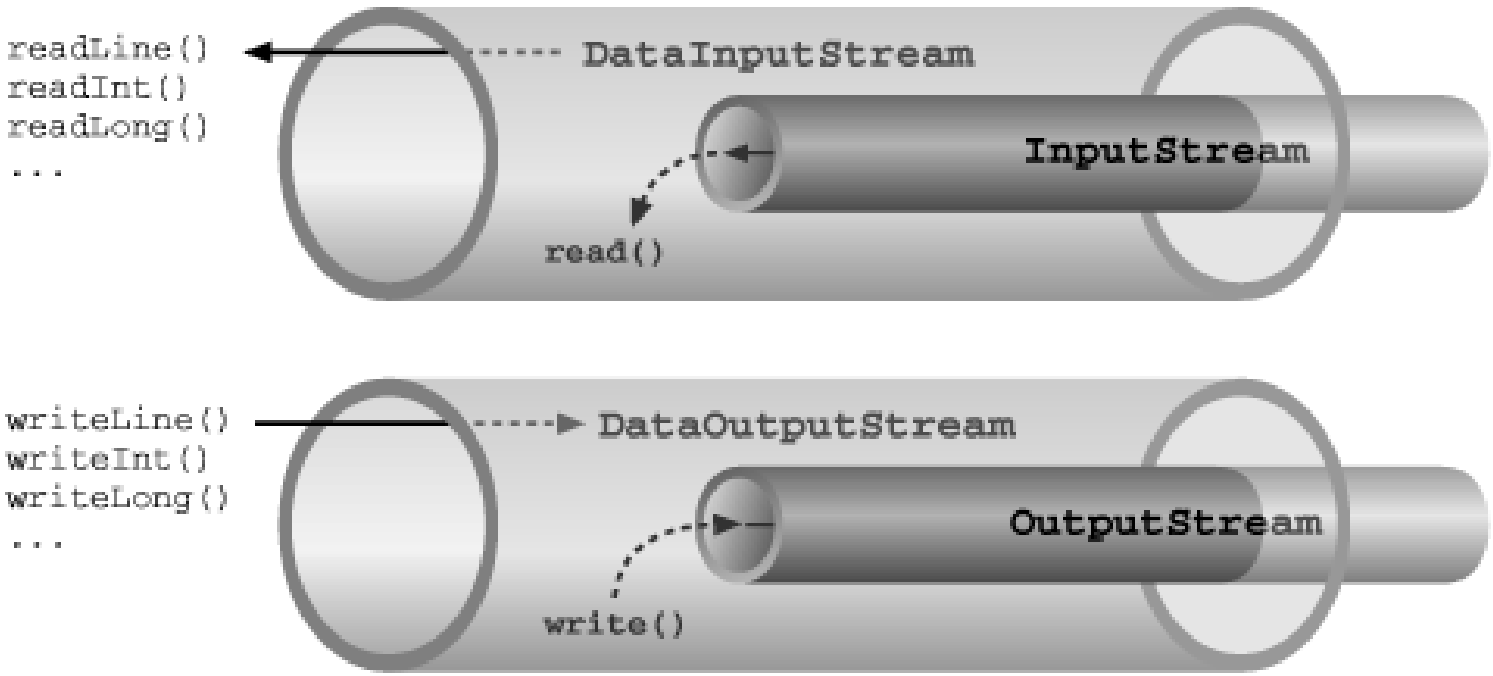
## Streams & File I/O

I/O Stream represents

- Input source
- Output destination

Streams support

- simple bytes
- primitive data types
- objects



# Summary

- Exceptions:
  - Handles disruptive runtime event
  - May handle event when encountered
  - Or earlier in execution path
  - Java Exception class derived from Throwable
  - Exceptions may be check or unchecked
  - Use only for exceptional conditions
  - Never use for ordinary control flow.

# Outstanding topics

## Java 8

This course based on Java 7.
Significant additions to Java language introduced in Java 8, March 2014

# Referenced Material

1. The Java Tutorials: Exceptions

`http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html`

[Accessed 2014-06-07]

2. Barnes David J., Kolling Michael. 2006. Third Edition. Objects First with Java. Ch. 12 Handling Errors

1. The Java Tutorials: Generic Types

`https://docs.oracle.com/javase/tutorial/java/generics/types.html`

[Accessed 2015-04-17]