INFO1113 Object-Oriented Programming

Week 9B: Inner Classes, Static Import and Javadoc

Slides developed by Tyson Thomas

Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Topics

- Inner Classes (s. 4)
  - Static Inner Class
  - Instance Inner Class
- Imports (s. 32)
- Javadocs (s. 36)

Inner Classes

Inner classes are classes that are defined within a class.

We have seen the usage of separate classes within the same file but we are able to scope classes within each other and grant access to the outer class.

Why?

Inner Classes

There are a lot of options we get from having enclosing classes.

- Access to outer class members
- Control the naming of classes
- Scoping classes
- Allow the outer class to utilise the inner class for its own purposes

Inner Classes

A common case where we may want to use an inner classes is where we have a similar concept but its definition may differ for each class.

- A common case is with linked data structures and the concept of a **Node**.

- **Iterators** that do not need to be exposed outside of their enclosing can be considered for being an inner class.

We should consider logical grouping of classes.

Inner Classes

Inner classes are referred to by their identifier within the class. We are able to apply regular encapsulation like other variables and control the scope of the variable.

In the event we want the class to be accessible outside of it, we can access it through the class if we specify the **public** access modifier.

Let’s look into “static” inner classes

“Static” Inner Class

A common confusion with the syntax is what is referred to as a static class within inside another class.

The static keyword that is applied is in fact on a property of the member variable.

It will act similar to a static variable of a class. We are able to access it without the need to instantiating an object.

“Static” Inner Class

Let’s consider the following code:

```
public class Truck {

    private static class Cargo {
        private String description;
        public Cargo(String description) {
            this.description = description;
        }

        public String getDescription() {
            return description;
        }

        public String toString() {
            return "[" + description + "]";
        }
    }
    private LinkedList<Cargo> cargo;
    public Truck() {
        cargo = new LinkedList<Cargo>();
    }
    //Snipped
    public void addCargo(String description) {
        cargo.add(new Cargo(description));
    }
}
```

```
public static void main(String[] args) {
        Truck t = new Truck();
        t.addCargo("Food");
        t.addCargo("CDs");
        t.addCargo("TVs");
        System.out.println(t);
    }
```

“Static” Inner Class

Let’s consider the following code:

```
public class Truck {

    private static class Cargo {
        private String description;
        public Cargo(String description) {
            this.description = description;
        }

        public String getDescription() {
            return description;
        }

        public String toString() {
            return "[" + description + "]";
        }
    }
    private LinkedList<Cargo> cargo;
    public Truck() {
        cargo = new LinkedList<Cargo>();
    }
    //Snipped
    public void addCargo(String description) {
        cargo.add(new Cargo(description));
    }
}
```

```
public static void main(String[] args) {
        Truck t = new Truck();
        t.addCargo("Food");
        t.addCargo("CDs");
        t.addCargo("TVs");
        System.out.println(t);
    }
```

We declare a **static class** within the **Truck** class.

“Static” Inner Class

Let’s consider the following code:

```
public class Truck {

    private static class Cargo {
        private String description;
        public Cargo(String description) {
            this.description = description;
        }

        public String getDescription() {
            return description;
        }

        public String toString() {
            return "[" + description + "]";
        }
    }
    private LinkedList<Cargo> cargo;
    public Truck() {
        cargo = new LinkedList<Cargo>();
    }
    //Snipped
    public void addCargo(String description) {
        cargo.add(new Cargo(description));
    }
}
```

```
public static void main(String[] args) {
        Truck t = new Truck();
        t.addCargo("Food");
        t.addCargo("CDs");
        t.addCargo("TVs");
        System.out.println(t);
    }
```

Just like any normal class we are able to define a constructor, fields and methods.

“Static” Inner Class

Let’s consider the following code:

```
public class Truck {

    private static class Cargo {
        private String description;
        public Cargo(String description) {
            this.description = description;
        }

        public String getDescription() {
            return description;
        }

        public String toString() {
            return "[" + description + "]";
        }
    }
    private LinkedList<Cargo> cargo;
    public Truck() {
        cargo = new LinkedList<Cargo>();
    }
    //Snipped
    public void addCargo(String description) {
        cargo.add(new Cargo(description));
    }
}
```

```
public static void main(String[] args) {
        Truck t = new Truck();
        t.addCargo("Food");
        t.addCargo("CDs");
        t.addCargo("TVs");
        System.out.println(t);
    }
```

This class cannot be accessed outside of the outer class due to its enclosure

What if were to mark with the public access modifier?

“Static” Inner Class

Let’s consider the following code:

```
public class Truck {

    public static class Cargo {
        private String description;
        public Cargo(String description) {
            this.description = description;
        }

        public String getDescription() {
            return description;
        }

        public String toString() {
            return "[" + description + "]";
        }
    }
    private LinkedList<Cargo> cargo;
    public Truck() {
        cargo = new LinkedList<Cargo>();
    }
    //Snipped
    public void addCargo(String description) {
        cargo.add(new Cargo(description));
    }
}
```

```
public static void main(String[] args) {
        Truck t = new Truck();
        t.addCargo("Food");
        t.addCargo("CDs");
        t.addCargo("TVs");
        System.out.println(t);
    }
```

To reiterate, we can specify class but how could we access it and use it outside of the **Truck** class?

“Static” Inner Class

Let’s consider the following code:

```
public class Truck {

    public static class Cargo {
        private String description;
        public Cargo(String description) {
            this.description = description;
        }

        public String getDescription() {
            return description;
        }

        public String toString() {
            return "[" + description + "]";
        }
    }
    private LinkedList<Cargo> cargo;
    public Truck() {
        cargo = new LinkedList<Cargo>();
    }
    //Snipped
    public void addCargo(String description) {
        cargo.add(new Cargo(description));
    }
}
```

```
public static void main(String[] args) {
        Truck t = new Truck();
        t.addCargo("Food");
        t.addCargo("CDs");
        t.addCargo("TVs");
        Truck.Cargo c = new Truck.Cargo(“Separate Cargo”);
        System.out.println(t);
    }
```

We can use the **inner class** definition within the **outer** class as a **type identifier.**

Same pattern with the constructor.

What about instance inner classes?

Inner classes

Similarly to static version we define by we do not apply the static access modifier.

However, there is a different relationship that the outer class can have with the inner class.

Inner classes

Non-static inner classes require an instance of the outer class to exist prior to instantiating the inner class. This inner class can have access to the outer class instance variables.

There is a clear coupled relationship between both instances. The inner class instance depends on the other instance existing once instantiated.

Inner classes

Let’s consider the following code:

```
public class Book implements Iterable<Book.Page> {

    public static class Page {
        public final String contents;
        public Page(String p) {
            contents = p;
        }
    }
    private class BookReader implements Iterator<Page> {
        private int index;
        public BookReader() { index = 0; }
        public boolean hasNext() { return index < pages.size(); }

        public Page next() {
            Page p = pages.get(index);
            index++;
            return p;
        }
    }
    private List<Page> pages;
    public Book() {
        pages = new ArrayList<Page>();
    }
    public void add(String contents) {
        pages.add(new Page(contents));
    }
    public Iterator<Book.Page> iterator() {
        return new BookReader();
    }
}
```

Inner classes

Let's consider the following code:

```
public class Book implements Iterable<Book.Page> {

    public static class Page {
        public final String contents;
        public Page(String p) {
            contents = p;
        }
    }
    private class BookReader implements Iterator<Page> {
        private int index;
        public BookReader() { index = 0; }
        public boolean hasNext() { return index < pages.size(); }

        public Page next() {
            Page p = pages.get(index);
            index++;
            return p;
        }
    }
    private List<Page> pages;
    public Book() {
        pages = new ArrayList<Page>();
    }
    public void add(String contents) {
        pages.add(new Page(contents));
    }
    public Iterator<Book.Page> iterator() {
        return new BookReader();
    }
}
```

We will be looking into an iterator example. We are using a **static inner type** as a **type argument** for **Iterable.**

Inner classes

Let’s consider the following code:

```
public class Book implements Iterable<Book.Page> {

    public static class Page {
        public final String contents;
        public Page(String p) {
            contents = p;
        }
    }
    private class BookReader implements Iterator<Page> {
        private int index;
        public BookReader() { index = 0; }
        public boolean hasNext() { return index < pages.size(); }

        public Page next() {
            Page p = pages.get(index);
            index++;
            return p;
        }
    }
    private List<Page> pages;
    public Book() {
        pages = new ArrayList<Page>();
    }
    public void add(String contents) {
        pages.add(new Page(contents));
    }
    public Iterator<Book.Page> iterator() {
        return new BookReader();
    }
}
```

We will be utilising a **inner class** that **implements** an **Iterator** of type **Page**.

Inner classes

Let’s consider the following code:

```
public class Book implements Iterable<Book.Page> {

    public static class Page {
        public final String contents;
        public Page(String p) {
            contents = p;
        }
    }
    private class BookReader implements Iterator<Page> {
        private int index;
        public BookReader() { index = 0; }
        public boolean hasNext() { return index < pages.size(); }

        public Page next() {
            Page p = pages.get(index);
            index++;
            return p;
        }
    }
    private List<Page> pages;
    public Book() {
        pages = new ArrayList<Page>();
    }
    public void add(String contents) {
        pages.add(new Page(contents));
    }
    public Iterator<Book.Page> iterator() {
        return new BookReader();
    }
}
```

We can see that the methods **hasNext** and **next** are able to access the **instance variables** in **Book**.

Inner classes

Let's consider the following code:

```
public class Book implements Iterable<Book.Page> {

    public static class Page {
        public final String contents;
        public Page(String p) {
            contents = p;
        }
    }
    private class BookReader implements Iterator<Page> {
        private int index;
        public BookReader() { index = 0; }
        public boolean hasNext() { return index < pages.size(); }

        public Page next() {
            Page p = pages.get(index);
            index++;
            return p;
        }
    }
    private List<Page> pages;
    public Book() {
        pages = new ArrayList<Page>();
    }
    public void add(String contents) {
        pages.add(new Page(contents));
    }
    public Iterator<Book.Page> iterator() {
        return new BookReader();
    }
}
```

```
public static void main(String[] args) {
    Book b = new Book();
    b.add("Line 1");
    b.add("Line 2");
    for(Book.Page p : b) {
        System.out.println(p.contents);
    }
}
```

We can see that the private inner class is able to be returned and refer to instance variables of **book**.

How would we use the type in our code?

Inner classes

Let’s consider the following code:

```
public class Book implements Iterable<Book.Page> {

    public static class Page {
        public final String contents;
        public Page(String p) {
            contents = p;
        }
    }
    public class BookReader implements Iterator<Page> {
        private int index;
        public BookReader() { index = 0; }
        public boolean hasNext() { return index < pages.size(); }

        public Page next() {
            Page p = pages.get(inde
            index++;
            return p;
        }
    }
    private List<Page> pages;
    public Book() {
        pages = new ArrayList<Page>();
    }
    public void add(String contents) {
        pages.add(new Page(contents));
    }
    public Iterator<Book.Page> iterator() {
        return new BookReader();
    }
}
```

```
public static void main(String[] args) {
    Book b = new Book();
    b.add("Line 1");
    b.add("Line 2");
    for(Book.Page p : b) {
        System.out.println(p.contents);
    }
}
```

Okay let’s change the access modifier to **public** and access it outside. Let’s also use a classic iterator pattern.

Inner classes

Let's consider the following code:

```
public class Book implements Iterable<Book.Page> {

    public static class Page {
        public final String contents;
        public Page(String p) {
            contents = p;
        }
    }
    public class BookReader implements Iterator<Page> {
        private int index;
        public BookReader() { index = 0; }
        public boolean hasNext() { return index < pages.size(); }

        public Page next() {
            Page p = pages.get(inde
            index++;
            return p;
        }
    }
    private List<Page> pages;
    public Book() {
        pages = new ArrayList<Page>();
    }
    public void add(String contents) {
        pages.add(new Page(contents));
    }
    public Iterator<Book.Page> iterator() {
        return new BookReader();
    }
}
```

```
public static void main(String[] args) {
    Book b = new Book();
    b.add("Line 1");
    b.add("Line 2");

    Book.BookReader reader = b.new BookReader();
    while(reader.hasNext()) {
        Book.Page p = reader.next();
        System.out.println(p.contents);
    }
}
```

Okay let's change the access modifier to **public** and access it outside. Let's also use a classic iterator pattern.

Since changing it to public we can access the class **but only through an instance of the outer class**.

The declaration type does not require an instance.

Inner classes

Let's consider the following code:

```
public class Book implements Iterable<Book.Page> {

    public static class Page {
        public final String contents;
        public Page(String p) {
            contents = p;
        }
    }
    public class BookReader implements Iterator<Page> {
        private int index;
        public BookReader() { index = 0; }
        public boolean hasNext() { return index < pages.size(); }

        public Page next() {
            Page p = pages.get(inde
            index++;
            return p;
        }
    }
    private List<Page> pages;
    public Book() {
        pages = new ArrayList<Page>();
    }
    public void add(String contents) {
        pages.add(new Page(contents));
    }
    public Iterator<Book.Page> iterator() {
        return new BookReader();
    }
}
```

```
public static void main(String[] args) {
    Book b = new Book();
    b.add("Line 1");
    b.add("Line 2");

    Book.BookReader reader = b.new BookReader();
    while(reader.hasNext()) {
        Book.Page p = reader.next();
        System.out.println(p.contents);
    }
}
```

Okay let's change the access modifier to **public** and access it outside. Let's also use a classic iterator pattern.

We are able to refer to it through its variable and use it as an iterator like before.

When would we use each one?

Inner Classes

Non-static inner class will utilise instance variables from the outer class.

Static inner class operate similarly to regular classes by their existence within another class is for grouping reasons.

Import

There are a couple of import methods within java.

As seen consistently throughout the semester, we have been able to utilise standard library classes by importing them and specifying the identifier or the wild card to import all classes.

```
import java.util.ArrayList;
import java.util.*;
import static java.lang.Math.PI;
import static java.lang.Math.*;
```

import

For the semester we have been using the import keyword to retrieve standard library classes.

There are two variants, we have commonly imported classed but using:

```
import static java.lang.Math.PI;
import static java.lang.Math.*;
```

We are able to import all static methods and variables accessible within the class and use them without referring to the class.

Importing

Let’s take a look at an example calculating magnitude

```
public static double magnitude(double v1, double v2) {
    return Math.sqrt((Math.pow(v1, 2) + Math.pow(v2, 2)));
}
```

Without the import we would need to specify the class each time to refer to the operation we want.

Importing

Let's take a look at an example calculating magnitude

```
public static double magnitude(double v1, double v2) {
    return Math.sqrt((Math.pow(v1, 2) + Math.pow(v2, 2)));
}
```

But this can be ugly to constantly write class name and we want to be more succinct with our methods,

```
import static java.lang.Math.*;

public static double magnitude(double v1, double v2) {
    return sqrt((pow(v1, 2) + pow(v2, 2)));
}
```

We are able to import all static variables and methods through **import static** and utilise them as if they were defined within the class.

Documenting our work

Javadoc

Documentation is an important aspect to application development. You are producing a technical manual for others to read so they can comprehend your code and utilise it.

Providing a solution is not enough, you will need to show how to use the solution.

https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

Javadoc

Large and complex methods can be very hard for anyone reading your code to understand. Working with any library/package produce by someone else can be difficult to understand.

You will commonly work in teams and produce code that has to be readable by others.

Javadoc

- Comments

Simply writing comments in your code allows you to always understand what a method/class is doing.

Complex methods sometimes require inline details so users unfamiliar with your library can understand what it is trying to do.

- Clear method names and style

Try to ensure you adhere to the library's style guide and when writing your method name, make it clear to whoever is reading it what the method performs.

So what’s javadoc?

Javadoc

Javadoc is a documentation generator for **Java**.

It extracts the **javadoc** comments for methods, fields and classes written within your java source files and produces a html documents.

The style is similar to the java api documentation.

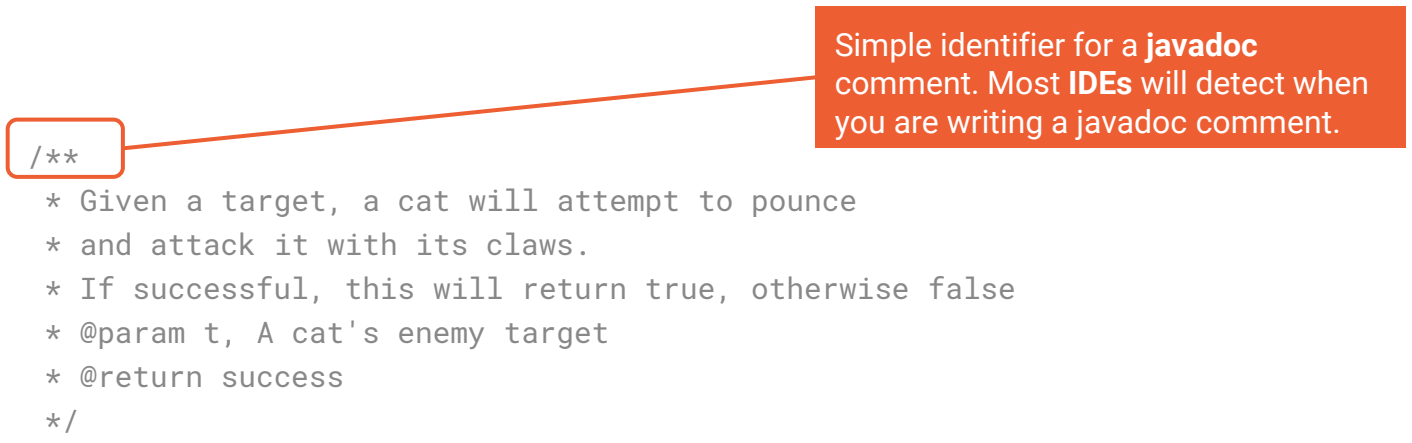
Javadoc

Javadoc comments have a simple identifier but can contain annotations that provide extra information about parameters and return types.

```
/**
 * Given a target, a cat will attempt to pounce
 * and attack it with its claws.
 * If successful, this will return true, otherwise false
 * @param t, A cat's enemy target
 * @return success
 */
```

Javadoc

Javadoc comments have a simple identifier but can contain annotations that provide extra information about parameters and return types.



Javadoc

Javadoc comments have a simple identifier but can contain annotations that provide extra information about parameters and return types.

```
/**
 * Given a target, a cat will attempt to pounce
 * and attack it with its claws.
 * If successful, this will return true, otherwise false
 * @param t, A cat's enemy target
 * @return success
 */
```

@param, allows us to specify a parameter, specifies a what the parameter is, allows explanation

Javadoc

Javadoc comments have a simple identifier but can contain annotations that provide extra information about parameters and return types.

```
/**
 * Given a target, a cat will attempt to pounce
 * and attack it with its claws.
 * If successful, this will return true, otherwise false
 * @param t, A cat's enemy target
 * @return success
 */
```

Similar to the @param annotation, @return outlines the return variable/value.

Javadoc

@param, parameter description, specified for as many parameters a method allows.

@see, reference annotation, useful for outlining issues with the method or what it has resolved.

@return, return type description. Specifies what is returned by the method and the conditions.

@since, when the method was included in your library

@throws, allows you to specify what exception it will throw and when it will throw it.

@deprecated, marks a method/class for deprecation (removal)

So how do we generate documentation?

Javadoc

Let’s consider the following class

```java
public class Cat {
    /**
     * The name of the cat
     */
    private String name;

    /**
     * Marks if the cat is in a playful state or not
     */
    private boolean playful;

    /**
     * New name counter, will be randomly assigned
     * when the name of the cat is changed
     */
    private int newNameCounter;

    /**
     * Old name of the cat
     */
    private String oldName;

    /**
     * Constructor for a cat, requires a name
     * default state for playful is false
     * @param name of the cat
     */
    public Cat(String name) {
        this.name = name;
        playful = false;
        newNameCounter = (int)(Math.random() * 30);
    }

    /**
     * Given a target, a cat will attempt to pounce
     * and attack it with its claws.
     * If successful, this will return true, otherwise false
     * @param t, A cat's enemy target
     * @return success
     */
    public boolean attack(Target t) {
        if(target.isRodent()) { return true; } else { return false; }
    }

    /**
     * if the newNameCounter greater than 0, old name is return
     * otherwise newName is returned.
     * @return oldName or name,
     */
    public String getName() {
        if(newNameCounter > 0) {
            return oldName;
        } else {
            newNameCounter--;
            return name;
        }
    }
}
```

Javadoc

Let's consider the following class

```java
public class Cat {
    /**
     * The name of the state
     */
    private String name;

    /**
     * Marks if the cat is in a playful state or not
     */
    private boolean playful;

    /**
     * New name counter, will be randomly assigned
     * when the name of the cat is changed
     */
    private int newNameCounter;

    /**
     * Old name of the cat
     */
    private String oldName;

    /**
     * Constructor for a cat, requires a name
     * default state for playful is false
     * @param name of the cat
     */
    public Cat(String name) {
        this.name = name;
        playful = false;
        newNameCounter = (int)(Math.random() * 30);
    }

    /**
     * Given a target, a cat will attempt to pounce
     * and attack it with its claws.
     * If successful, this will return true, otherwise false
     * @param t, A cat's enemy target
     * @return success
     */
    public boolean attack(Target t) {
        if(target.isRodent()) { return true; } else { return false; }
    }

    /**
     * if the newNameCounter greater than 0, old name is return
     * otherwise newName is returned.
     * @return oldName or name,
     */
    public String getName() {
        if(newNameCounter > 0) {
            return oldName;
        } else {
            newNameCounter--;
            return name;
        }
    }
}
```

Specified a javadoc comment for a constructor, showing the @param name.

Javadoc

Let's consider the following class

```java
public class Cat {
    /**
     * The name of the state
     */
    private String name;

    /**
     * Marks if the cat is in a playful state or not
     */
    private boolean playful;

    /**
     * New name counter, will be randomly assigned
     * when the name of the cat is changed
     */
    private int newNameCounter;

    /**
     * Old name of the cat
     */
    private String oldName;

    /**
     * Constructor for a cat, requires a name
     * default state for playful is false
     * @param name of the cat
     */
    public Cat(String name) {
        this.name = name;
        playful = false;
        newNameCounter = (int)(Math.random() * 30);
    }

    /**
     * Given a target, a cat will attempt to pounce
     * and attack it with its claws.
     * If successful, this will return true, otherwise false
     * @param t, A cat's enemy target
     * @return success
     */
    public boolean attack(Target t) {
        if(target.isRodent()) { return true; } else { return false; }
    }

    /**
     * if the newNameCounter greater than 0, old name is return
     * otherwise newName is returned.
     * @return oldName or name,
     */
    public String getName() {
        if(newNameCounter > 0) {
            return oldName;
        } else {
            newNameCounter--;
            return name;
        }
    }
}
```

We specify the parameter and a return description.

Javadoc

The javadoc command line tool requires us to specify a directory where the documents will be generated and point to our source files.

```
> javadoc -d <directory> <path to .java files>
```

Javadoc

Directories containing subpackages require the use of the -subpackages flag to recursively search all .java files and map each package in the document.

```
> javadoc -d <directory> -subpackages packageName
```

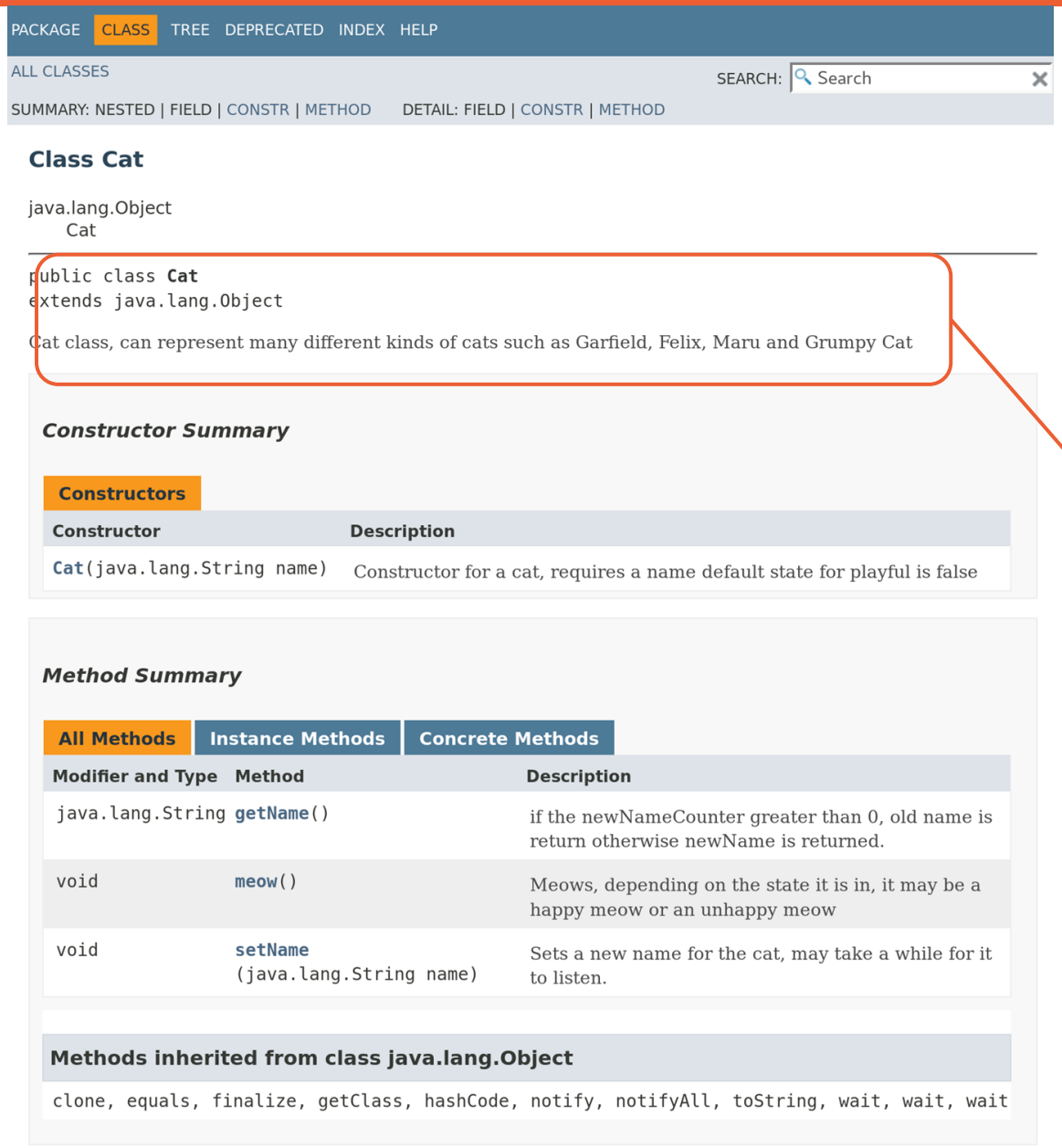
Javadoc

Output of the following command will produce a directory called **cat_docs** which will contain the generated documentation files.

```
> javadoc -d cat_docs Cat.java
```

./cat_docs

Javadoc



PACKAGE CLASS TREE DEPRECATED INDEX HELP

ALL CLASSES SEARCH: Search

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Cat

java.lang.Object
Cat

```
public class Cat
extends java.lang.Object
```

Cat class, can represent many different kinds of cats such as Garfield, Felix, Maru and Grumpy Cat

Constructor Summary

Constructors

Constructor	Description
`Cat(java.lang.String name)`	Constructor for a cat, requires a name default state for playful is false

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
java.lang.String	getName()	if the newNameCounter greater than 0, old name is return otherwise newName is returned.
void	meow()	Meows, depending on the state it is in, it may be a happy meow or an unhappy meow
void	setName (java.lang.String name)	Sets a new name for the cat, may take a while for it to listen.

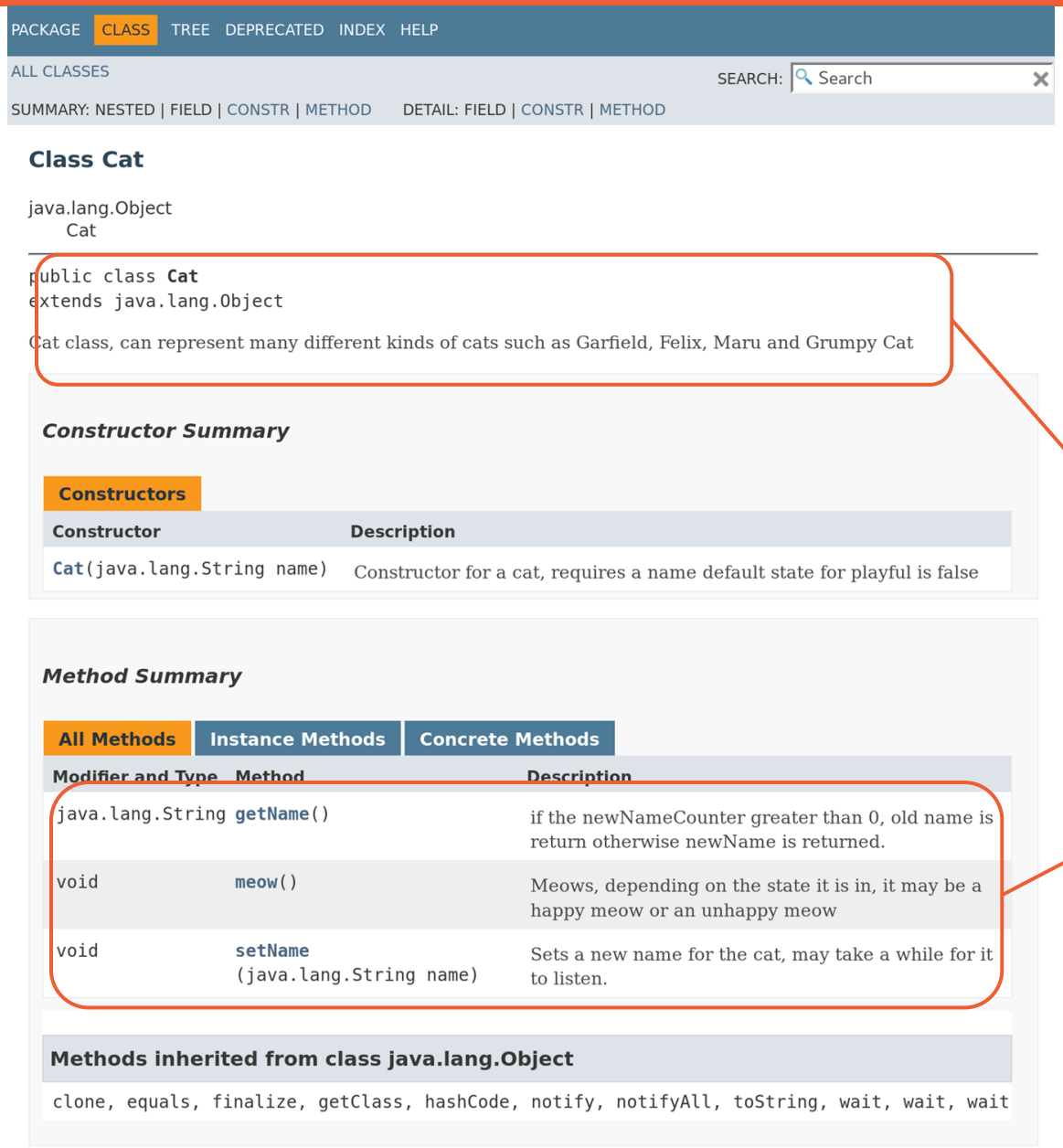
Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

We have produced our Cat class documentation.

Showing the documentation above the class.

Javadoc



PACKAGE CLASS TREE DEPRECATED INDEX HELP

ALL CLASSES SEARCH: Search

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Cat

java.lang.Object
Cat

```
public class Cat
extends java.lang.Object
```

Cat class, can represent many different kinds of cats such as Garfield, Felix, Maru and Grumpy Cat

Constructor Summary

Constructors

Constructor	Description
`Cat(java.lang.String name)`	Constructor for a cat, requires a name default state for playful is false

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
`java.lang.String`	`getName()`	if the newNameCounter greater than 0, old name is return otherwise newName is returned.
`void`	`meow()`	Meows, depending on the state it is in, it may be a happy meow or an unhappy meow
`void`	`setName (java.lang.String name)`	Sets a new name for the cat, may take a while for it to listen.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

We have produced our Cat class documentation.

Showing the documentation above the class.

The documentation for each method and their description.

Let’s go through the example

See you next time!