PGR112 – Step 7: Static, final and inheritance

Object Oriented Programming

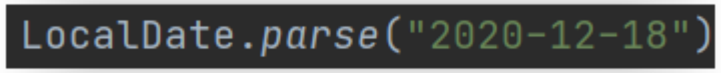
Bogdan Marculescu/ bogdan.marculescu@kristiania.no

Agenda

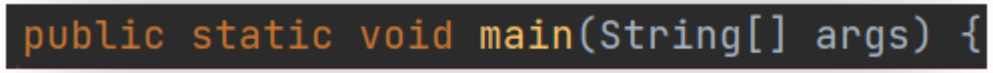
- Static
- Final
- Inheritance

Static

- We have seen this before:

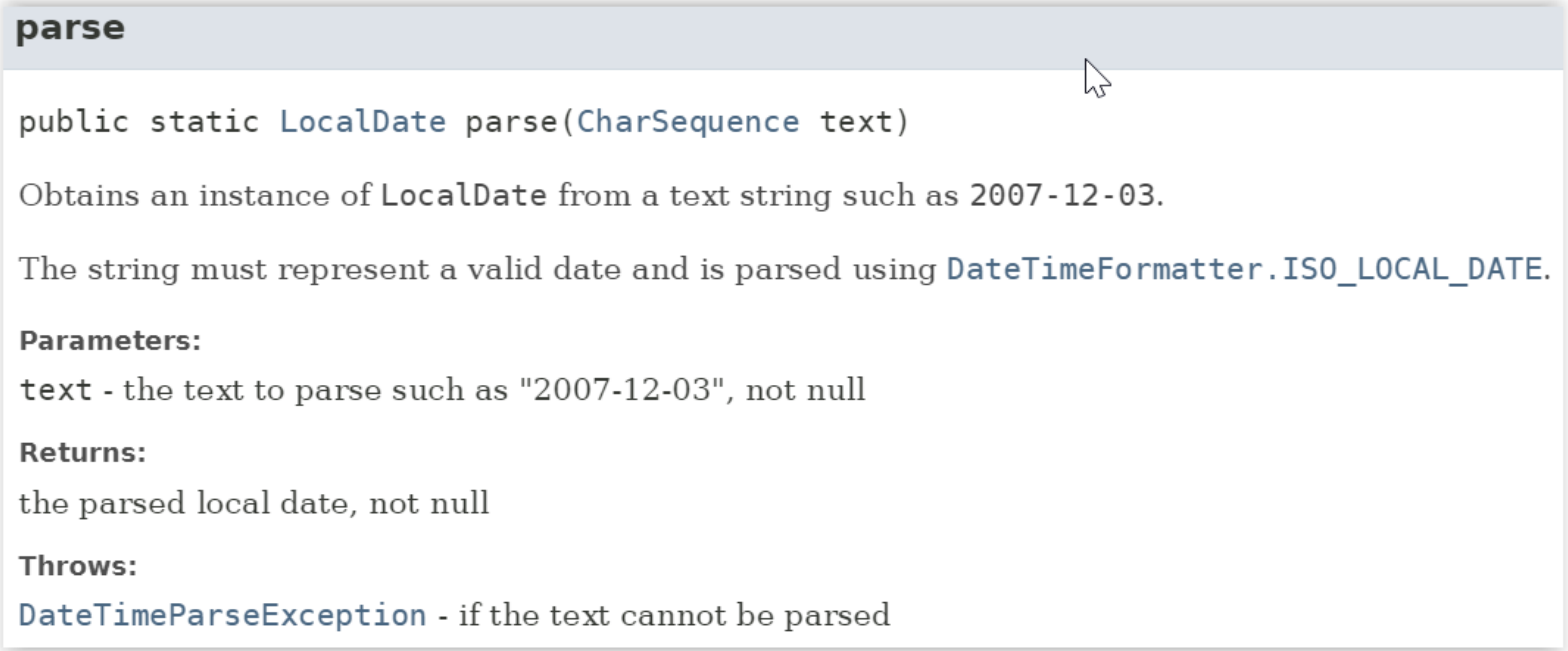


```
LocalDate.parse("2020-12-18")
```



```
public static void main(String[] args) {
```

LocalDate.parse()



parse

```
public static LocalDate parse(CharSequence text)
```

Obtains an instance of `LocalDate` from a text string such as `2007-12-03`.

The string must represent a valid date and is parsed using `DateTimeFormatter.ISO_LOCAL_DATE`.

Parameters:

`text` - the text to parse such as "2007-12-03", not null

Returns:

the parsed local date, not null

Throws:

`DateTimeParseException` - if the text cannot be parsed

Java API (15)

Static methods

- So we can have static methods in classes.
- This is in contrast to having methods in objects.
- We can call on the method without having an object of class.
- The syntax is a little different than what we are used to.

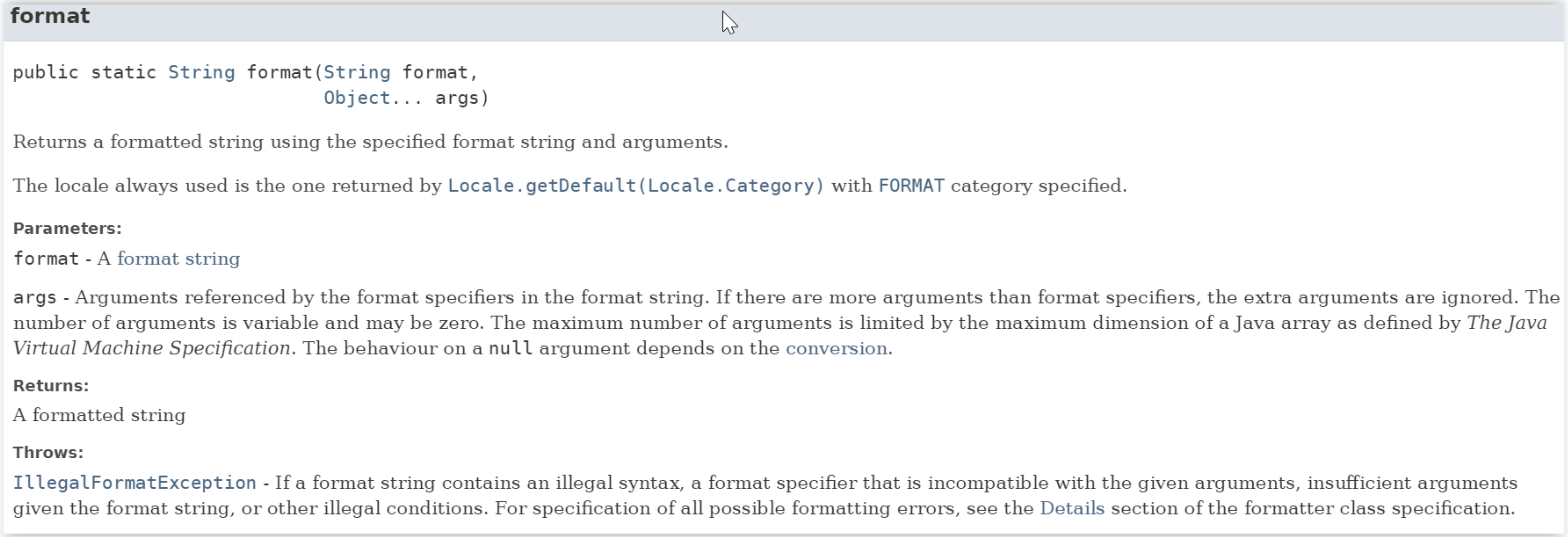
- Regular (dynamic) method:

```
Person p = new Person();
p.someMethod();
```

- Static method

```
LocalDate date = LocalDate.parse("1974-01-21");
```

More examples (String)


format

```
public static String format(String format,
                            Object... args)
```

Returns a formatted string using the specified format string and arguments.

The locale always used is the one returned by `Locale.getDefault(Locale.Category)` with `FORMAT` category specified.

Parameters:

`format` - A format string

`args` - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by *The Java Virtual Machine Specification*. The behaviour on a `null` argument depends on the conversion.

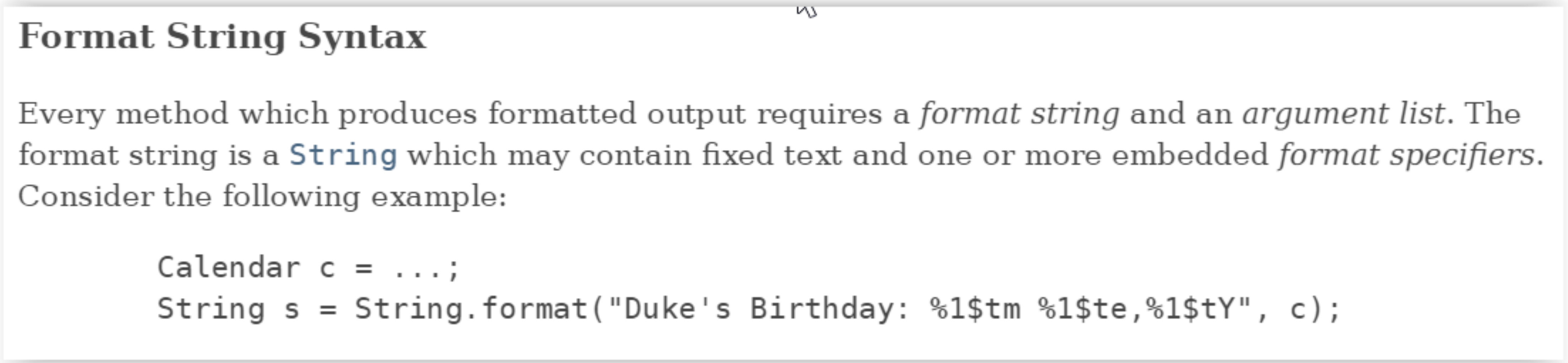
Returns:

A formatted string

Throws:

`IllegalFormatException` - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the Details section of the formatter class specification.

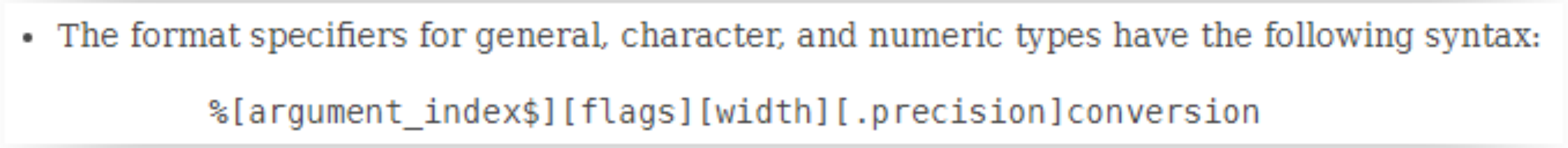
Format string



Format String Syntax

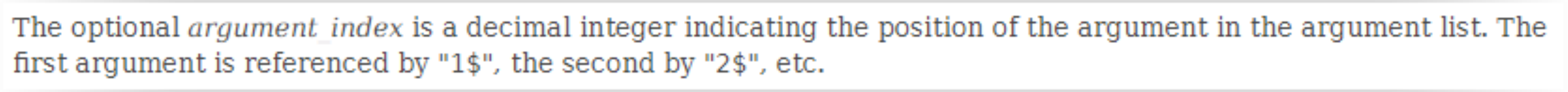
Every method which produces formatted output requires a *format string* and an *argument list*. The format string is a `String` which may contain fixed text and one or more embedded *format specifiers*. Consider the following example:

```
Calendar c = ...;
String s = String.format("Duke's Birthday: %1$tm %1$te,%1$tY", c);
```



- The format specifiers for general, character, and numeric types have the following syntax:

```
%[argument_index$][flags][width][.precision]conversion
```



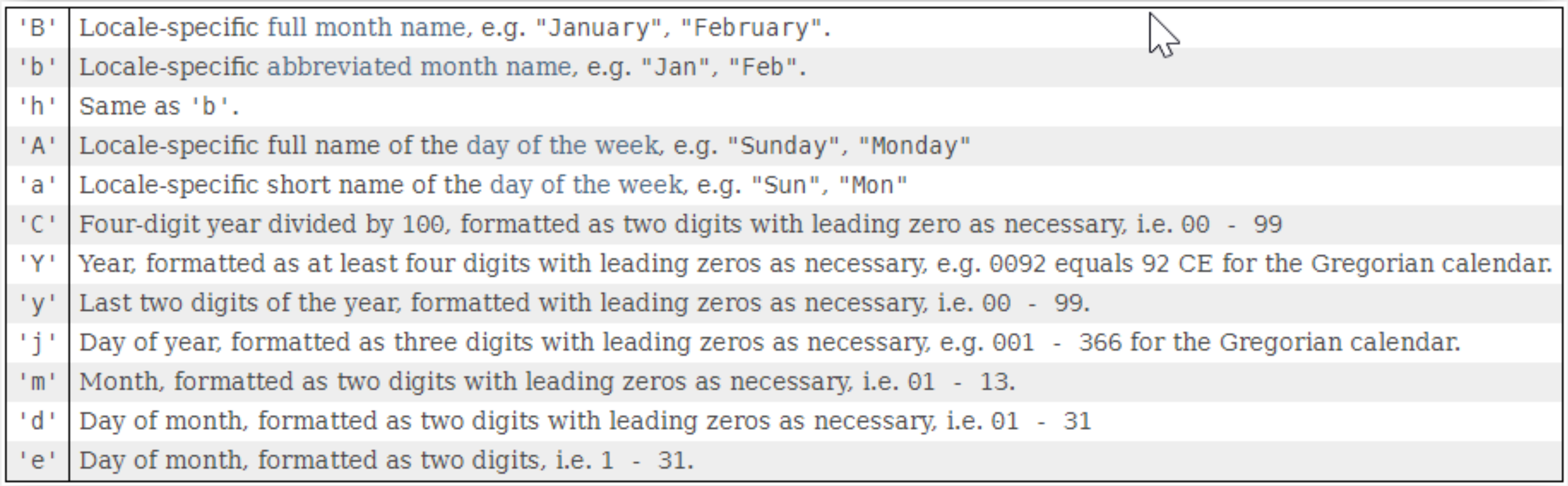
The optional *argument_index* is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by "1$", the second by "2$", etc.

Java Doc

Conversions

- s / S: Textual representation. (n gives line breaks - ie% n)
- t / T: This is about date or time

- And when it comes to dates:



'B'	Locale-specific full month name, e.g. "January", "February".
'b'	Locale-specific abbreviated month name, e.g. "Jan", "Feb".
'h'	Same as 'b'.
'A'	Locale-specific full name of the day of the week, e.g. "Sunday", "Monday"
'a'	Locale-specific short name of the day of the week, e.g. "Sun", "Mon"
'C'	Four-digit year divided by 100, formatted as two digits with leading zero as necessary, i.e. 00 - 99
'Y'	Year, formatted as at least four digits with leading zeros as necessary, e.g. 0092 equals 92 CE for the Gregorian calendar.
'y'	Last two digits of the year, formatted with leading zeros as necessary, i.e. 00 - 99.
'j'	Day of year, formatted as three digits with leading zeros as necessary, e.g. 001 - 366 for the Gregorian calendar.
'm'	Month, formatted as two digits with leading zeros as necessary, i.e. 01 - 13.
'd'	Day of month, formatted as two digits with leading zeros as necessary, i.e. 01 - 31
'e'	Day of month, formatted as two digits, i.e. 1 - 31.

Demo – String.format()

Your own static methods

- We can choose to create our own static methods.
- They can not be dependent on fields - ie variables in objects. These are class methods…

When should we have our own static methods?

- To access or manipulate static variables.
- Methods that do not depend on the state of an object. As we have seen examples of:
  - LocalDate.now ()
  - LocalDate.parse (…)
  - String.format (…)

Static variables

- We can choose to have static variables. They are often called class variables, because they are common to all objects of a class.

Some common static variables

- Math.PI –the number π.
- Color.BLACK – the color BLACK.

These might be useful later…

When do we used our own static (class) variables?

- When the value is not associated with objects of the class.
- When we want a common value for all objects of the class.

- In practice, static variables are mostly used as constants. Example:

```
public static final String COMPANY_NAME = "Telenor";
```

final

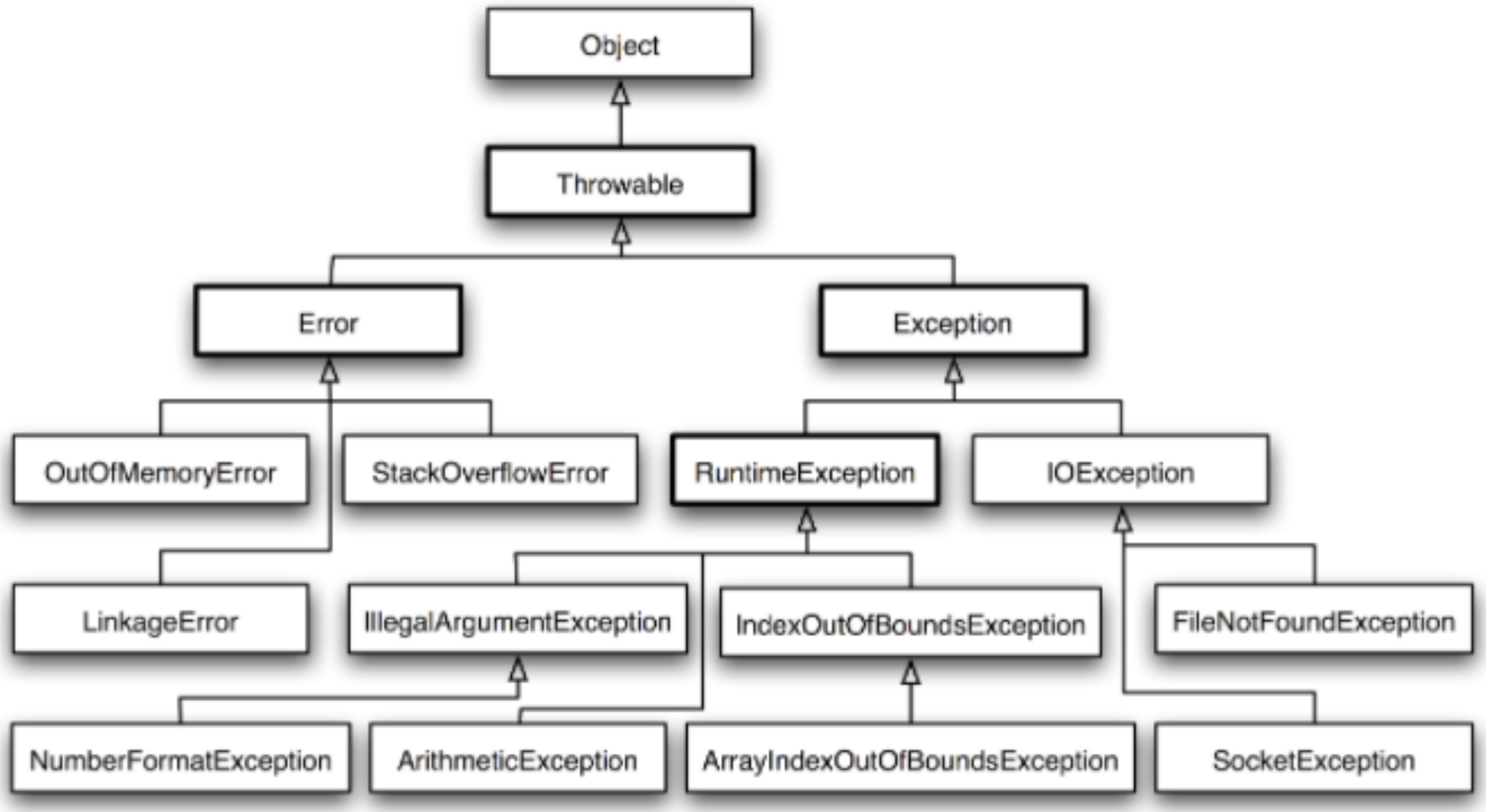
- Means that something can not be changed.
- We can have the final
  - Variables -> Constants (remember to write the variable name in UPPERCASE)
  - Methods -> Can not be overridden
  - Classes -> Can not be inherited

Inheritance!

- Classes can be (are?) related☺
- Let's look at an example from the world of Exceptions…

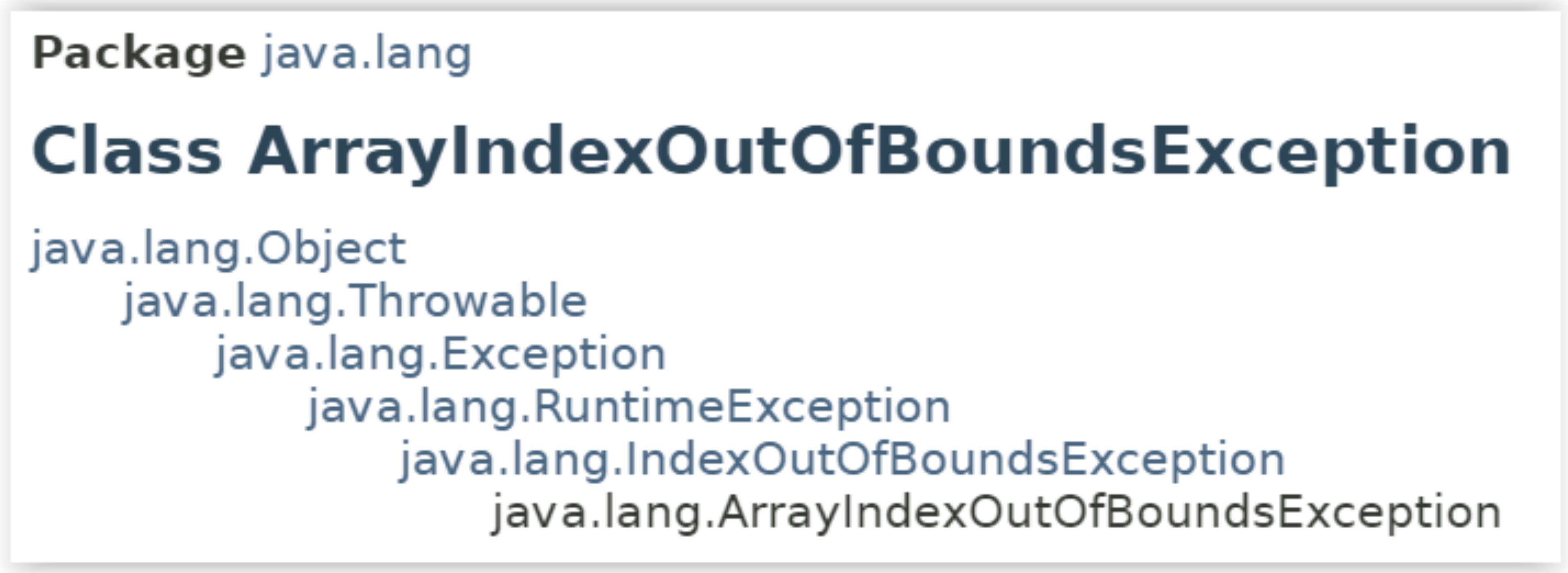
Exceptions

Figure 7. A Sample of Exception Types



protechtraining

ArrayIndexOutOfBoundsException



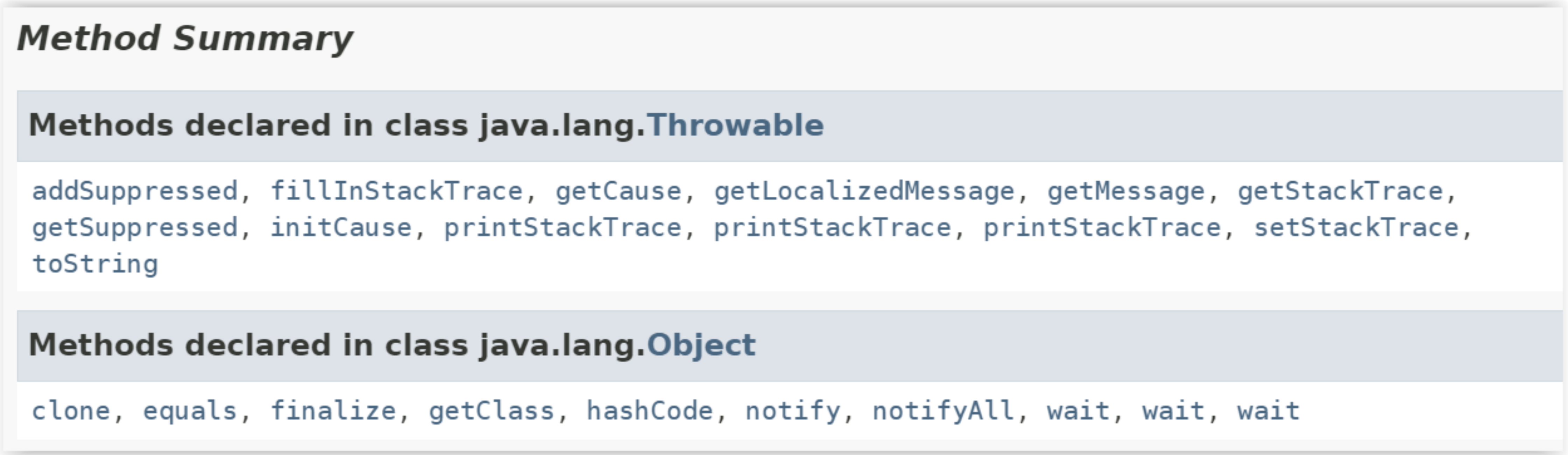
Package java.lang

Class ArrayIndexOutOfBoundsException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IndexOutOfBoundsException
                    java.lang.ArrayIndexOutOfBoundsException
```

[Java API (15)]

Methods in ArrayIndexOutOfBoundsException



Method Summary

Methods declared in class java.lang.Throwable

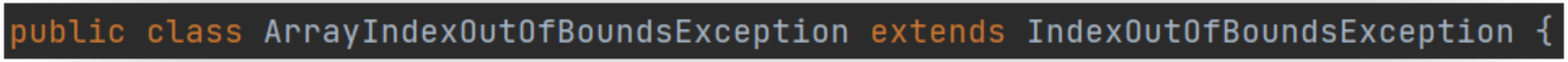
addSuppressed, fillInStackTrace, getCause, getLocalizedMessage, getMessage, getStackTrace, getSuppressed, initCause, printStackTrace, printStackTrace, printStackTrace, setStackTrace, toString

Methods declared in class java.lang.Object

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

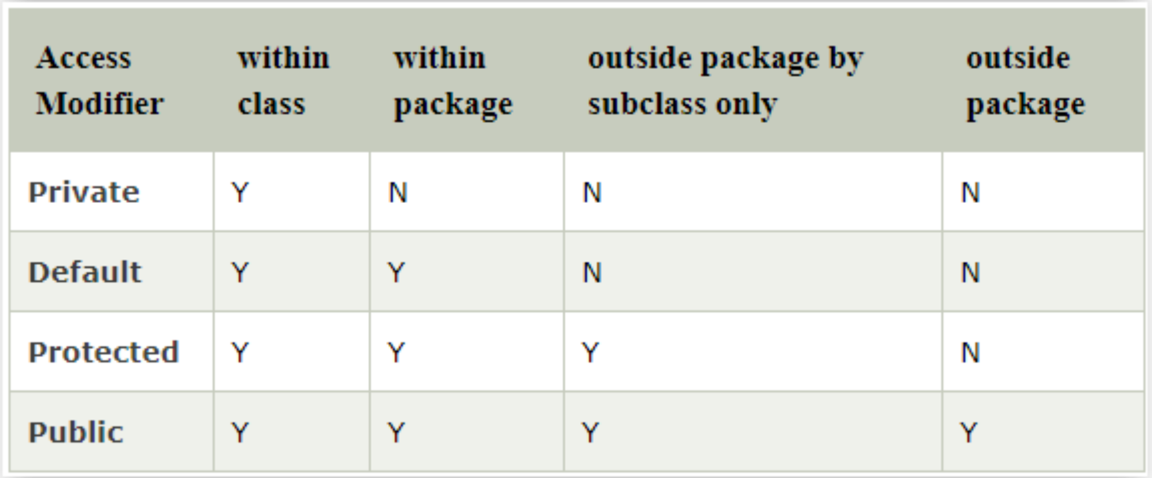
How do we use inheritance?

- We use «extends». Example:



```
public class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException {
```

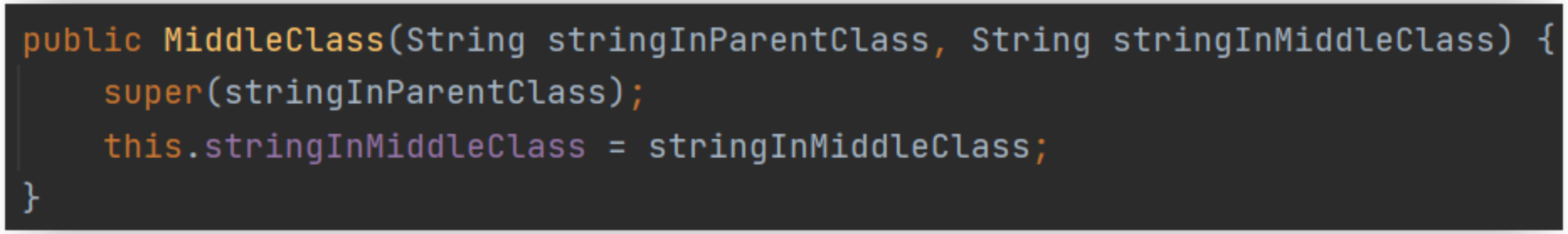
- The class inherits fields and methods from the parent.
- Cannot inherit from final class.
- Does not have access to private fields and methods.



Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Constructors and Inheritance

- In a constructor we can call the constructor of the superclass using the method super(<relevant arguments>)
- Examlpe (consider the class MiddleClass):



```java
public MiddleClass(String stringInParentClass, String stringInMiddleClass) {
    super(stringInParentClass);
    this.stringInMiddleClass = stringInMiddleClass;
}
```

- Note! When calling the superclass constructor, it MUST be the first line of code in the constructor.
- Note2! If you do NOT call super () in the constructor, the superclass default constructor will be called (the empty constructor). If it does not exist, you will not be able to compile.

Demo – inheritance

Method overriding

- We can override the methods we inherit. So we can change how they behave – to make them relevant to our new object.

Some rules for overriding

- To reuse the superclass method, use the method name super(args).
- When you override a method, you can give more access than the superclass (not less). Ex:
  - Superclass: protected void methodName () -> Subclass: public void methodName ()
- Can not override final methods (as I mentioned earlier today)
- We can not override static methods (but we can «at home» them).
- We can not override private methods (we do not have access to them)
- The method we create as overrides must return the same type (or a sub-type).
  - Superclass: protected Exception methodName () -> Subclass: protected RuntimeException methodName ()

Some rules that have to do with exceptions

- If the override method does not throw an exception, then the override method cannot throw the Checked exception.
  - protected void methodName () -> ~~protected void methodName () throws FileNotFoundException~~
- If the override method throws an exception, then the overriding method cannot throw a parent-exception (but can throw the same or sub-exception or no exception)
  - protected void methodName () throws FileNotFoundException->
    ~~protected void methodName () throws IOException~~

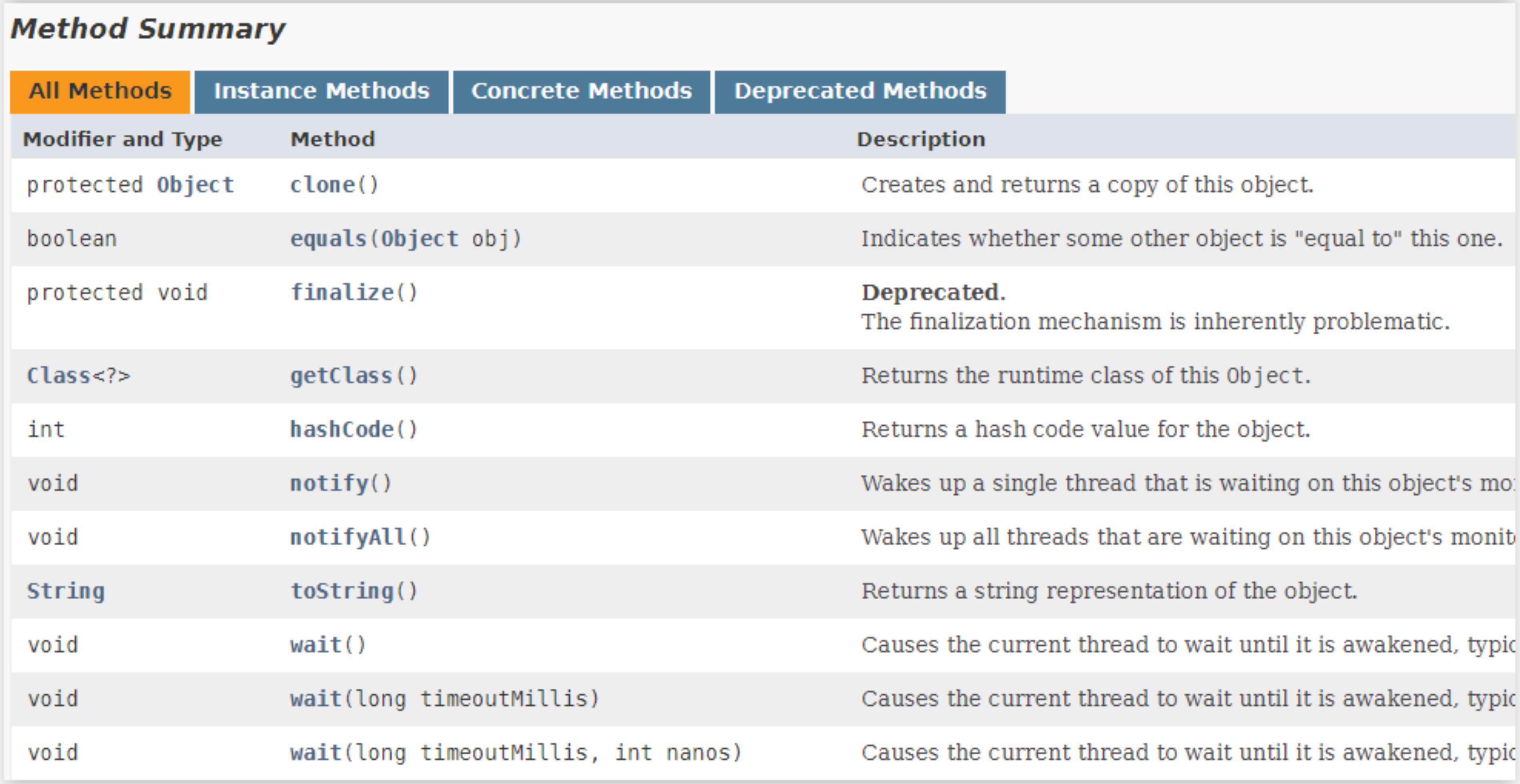
@Override

- When we override a method, we can choose to use an annotation: @Override
- Annotations are information about our code (metadata).
- In this case, we tell that we are overriding a method. If it turns out that we are not actually overriding a method, then the compiler will say that something is wrong: "You say that you are overriding, but you actually do not".
- It is therefore recommended to use @Override when we intend to override a method. It also increases readability (easier to see what is happening in the code).

Object

- We saw that at the very top of the ArrayIndexOutOfBoundsException family tree was Object.
- This is the root class of all classes.
- Geez, so all classes inherit methods from Object?
- Yes! But these may have been overridden in the family tree…

Methods in Object



Method Summary

All Methods | Instance Methods | Concrete Methods | Deprecated Methods

Modifier and Type	Method	Description
`protected Object`	`clone()`	Creates and returns a copy of this object.
`boolean`	`equals(Object obj)`	Indicates whether some other object is "equal to" this one.
`protected void`	`finalize()`	**Deprecated.** The finalization mechanism is inherently problematic.
`Class<?>`	`getClass()`	Returns the runtime class of this `Object`.
`int`	`hashCode()`	Returns a hash code value for the object.
`void`	`notify()`	Wakes up a single thread that is waiting on this object's mo
`void`	`notifyAll()`	Wakes up all threads that are waiting on this object's monit
`String`	`toString()`	Returns a string representation of the object.
`void`	`wait()`	Causes the current thread to wait until it is awakened, typic
`void`	`wait(long timeoutMillis)`	Causes the current thread to wait until it is awakened, typic
`void`	`wait(long timeoutMillis, int nanos)`	Causes the current thread to wait until it is awakened, typic

Java API (15)

toString

- That is a method in Object that all objects therefore inherit.
- Textual representation of an object.

When should/can we use inheritance?

- "Is-a" -relationship (not "has-a").
- An ArrayIndexOutOfBoundsException is an IndexOutOfBoundsException which is a RuntimeException etc.
- Some other classic examples:
  - MountainBike is a Bicycle
  - Cookbook is a Book (from the syllabus book)
  - Computer programmer is an Employee
  - Apple is a Fruit
- We use inheritance with caution. It is an advantage to be able to inherit traits. But a hereditary relationship gives a strong connection to another class. And we can then not inherit to other classes (multiple inheritance).

In summary

- Learning objectives of the week:
  - I can create a class that inherits from another class.
  - I know how to override a method.
  - I understand what static means for fields and methods.
  - I understand what the final entails.
  - I know the Object class and some of the methods that lie there.