INFO1113 Object-Oriented Programming

Week 5B: Overloading and how inheritance is applied

Slides developed by Tyson Thomas

Copyright Warning

Topics

- Method Overloading (s. 4)
- Constructor Overloading (s. 22)
- A peak at java.io (s. 37)
- **try-catch and exceptions** (s. 45)

Overloading

Firstly! What is **overloading?**

In regards to **Java** we are able to use the same method name but with different method **signature**.

Simply:

We are able to define a **method** such as **add** and have a version that accepts two integers and another version that accepts three integers.

Overloading

Firstly! What is **overloading?**

In regards to **Java** we are able to use the same method name but with different method **signature**.

Simply:

We are able to define a **method** such as **add** and have a version that accepts two integers and another version that accepts three integers.

```
int add(int a, int b)

int add(int a, int b, int c)
```

Overloading

Firstly! What is **overloading?**

In regards to **Java** we are able to use the same method name but with different method **signature**.

Simply:

We are able to define a **method** such as **add** and have a version that accepts two integers and another version that accepts three integers.

```
int add(int a, int b)
int add(int a, int b, int c)
```

Same name but both have different parameters, therefore different signature.

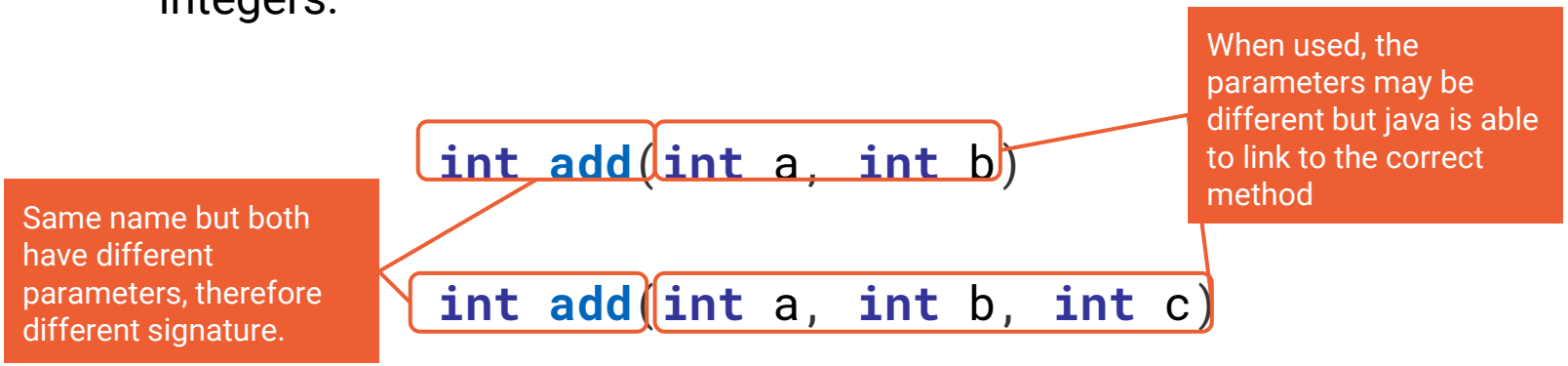
Overloading

Firstly! What is **overloading?**

In regards to **Java** we are able to use the same method name but with different method **signature**.

Simply:

We are able to define a **method** such as **add** and have a version that accepts two integers and another version that accepts three integers.



Where it is invalid

We are unable to apply overloading if we have a different **return type** between the methods. The return type is not part of the method signature.

For example:

```
float[] crossProduct(float[] a, float[] b)

int[] crossProduct(float[] a, float[] b)
```

Where it is invalid

We are unable to apply overloading if we have a different **return type** between the methods. The return type is not part of the method signature.

For example:

```
float[] crossProduct(float[] a, float[] b)

int[] crossProduct(float[] a, float[] b)
```

Even though float[] and int[] are specified here, the compile cannot specify which method it will call.

Hang on, but we have type information there!

Where it is invalid

We are unable to apply overloading if we have a different **return type** between the methods. The return type is not part of the method signature.

For example:

True! We do have type info there! But as we learned from the previous lecture, all classes inherit from Object!

```
float[] crossProduct(float[] a, float[] b)

int[] crossProduct(float[] a, float[] b)
```

Even though float[] and int[] are specified here, the compile cannot specify which method it will call.

Where it is invalid

So let's consider the following method calls using the two methods and assume that they are correct.

```
float[] crossProduct(float[] a, float[] b)
int[] crossProduct(float[] a, float[] b)
```

Method calls:

```
int[] x = crossProduct(null, null);

float[] y = crossProduct(null, null);

Object o = crossProduct(null, null);
```

Where it is invalid

So let's consider the following method calls using the two methods and assume that they are correct.

```
float[] crossProduct(float[] a, float[] b)
int[] crossProduct(float[] a, float[] b)
```

Method calls:

```
int[] x = crossProduct(null, null);

float[] y = crossProduct(null, null);

Object o = crossProduct(null, null);
```

Remember we can input null here! So, if the return type is different between two method, how does it know which one to return?

Where it is invalid

So let's consider the following method calls using the two methods and assume that they are correct.

```
float[] crossProduct(float[] a, float[] b)
int[] crossProduct(float[] a, float[] b)
```

Method calls:

```
int[] x = crossProduct(null, null);

float[] y = crossProduct(null, null);

Object o = crossProduct(null, null);
```

Remember we can input null here! So, if the return type is different between two method, how does it know which one to return?

But we have variable assignment information! Why can't it use this?

Where it is invalid

So let's consider the following method calls using the two methods and assume that they are correct.

```
float[] crossProduct(float[] a, float[] b)
int[] crossProduct(float[] a, float[] b)
```

Method calls:

```
int[] x = crossProduct(null, null);

float[] y = crossProduct(null, null);

Object o = crossProduct(null, null);
```

Remember we can input null here! So, if the return type is different between two method, how does it know which one to return?

But we have variable assignment information! Why can't it use this?

Oh…
Methods don't know what they are being assigned to and Object type is valid assignment type,

What about some ambiguous scenarios?

Ambiguous scenario

So let's consider the following method calls using the two methods and assume that they are correct.

```
int[] crossProduct(int[] a, int[] b)
int[] crossProduct(float[] a, float[] b)
```

Method calls:

```
int[] x = crossProduct(null, null);

int[] y = crossProduct(null, null);
```

Which method could it be calling?

Ambiguous scenario

So let's consider the following method calls using the two methods and assume that they are correct.

```
int[] crossProduct(int[] a, int[] b)
int[] crossProduct(float[] a, float[] b)
```

Method calls:

```
int[] x = crossProduct(null, null);

int[] y = crossProduct(null, null);
```

Which method could it be calling?

```
> javac OverloadTest.java
OverloadTest.java:15: error: reference to crossProduct is
ambiguous
            int[] o = crossProduct(null, null);
                      ^
  both method crossProduct(float[],float[]) in OverloadTest
and method crossProduct(int[],int[]) in OverloadTest match
1 error
```

The compiler would be unable to determine exactly what method is trying to be called and will throw an error.

Ambiguous scenario

So let's consider the following method calls using the two methods and assume that they are correct.

```
int[] crossProduct(int[] a, int[] b)
int[] crossProduct(float[] a, float[] b)
```

Method calls:

```
int[] x = crossProduct((int[])null, (int[])null);

int[] y = crossProduct((float[])null, (float[])null);
```

By casting the reference to a certain type, the compiler can deduce what method to call

By casting float[] on the null references we can see it infer the method with floats as arguments

If we want to deal with ambiguous statements like this, we will need to cast, considering the **Object** example from before, we could actually be passing a real reference to an array

So let’s demo this!

Constructor Overloading

We can observe the same overloading concept applied to constructors. This can be applied to both overloaded constructors within the same class as well as super constructors.

We are able to also utilise certain constructors for other constructors if we have already defined that behaviour.

Constructor Overloading

Let’s take a look at the following class

```
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        name = "Jeff";
        age = DEFAULT_AGE;
    }

    public Person(String name) {
        this.name = name;
        this.age = DEFAULT_AGE;
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Constructor Overloading

Let's take a look at the following class

```
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        name = "Jeff";
        age = DEFAULT_AGE;
    }

    public Person(String name) {
        this.name = name;
        this.age = DEFAULT_AGE;
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

We can see that there are 3 different constructors. Within our own code we choose to call anyone, in fact we have already been doing this!

Constructor Overloading

Let's take a look at the following class

```
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        name = "Jeff";
        age = DEFAULT_AGE;
    }

    public Person(String name) {
        this.name = name;
```

We can see that there are 3 different constructors. Within our own code we

```
public static void main(String[] args) {
    Person p1 = new Person(); //Jeff the default human!
    Person p2 = new Person("Janice");
    Person p3 = new Person("Dave", 32);
}
```

Since each constructor has a unique signature, we are able to utilise specific constructors by satisfying the correct types.

`this` keyword

The **`this`** keyword can play an important role in regards to constructors. It allows us to refer to the constructor within the context of a class.

In particular, we can reduce the amount of code we write by reusing a constructor.

Constructor Overloading

How could we use the this keyword in this example?

```java
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        name = "Jeff";
        age = DEFAULT_AGE;
    }

    public Person(String name) {
        this.name = name;
        this.age = DEFAULT_AGE;
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```java
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        this("Jeff", DEFAULT_AGE);
    }

    public Person(String name) {
        this(name, DEFAULT_AGE);
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Constructor Overloading

How could we use the this keyword in this example?

```
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        this("Jeff", DEFAULT_AGE);
    }

    public Person(String name) {
        this(name, DEFAULT_AGE);
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Calls this constructor

By using the **this** keyword, we are able to eliminate 2 lines from the other constructors by using the last one.

`super` keyword

We saw the use of the **super** keyword. We will be exploring inheritance with constructor overloading and method overriding and how we are able to utilise inherited behaviour in our program.

We will show how we are able to access elements through the **super** keyword.

So how does it work with inheritance?

We saw last lecture that we could specify the constructor we want to use and using the **super** keyword.

Let’s bring in the **Employee** class to inherit from **Person.**

Constructor Overloading

```java
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        this("Jeff", DEFAULT_AGE);
    }

    public Person(String name) {
        this(name, DEFAULT_AGE);
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    //<snipped getName(), setName(), getAge()
}
```

```java
public class Employee extends Person {

    private long employeeId;
    private long departmentId;

    public Employee(String name, int age,
        long departmentId, long employeeId) {

        super(name, age);
        this.departmentId = departmentId;
        this.employeeId = employeeId;
    }
    //<snipped other methods
}
```

Constructor Overloading

```java
public class Person {

    private static int DEFAULT_AGE = 21;

    private String name;
    private int age;

    public Person() {
        this("Jeff", DEFAULT_AGE);
    }

    public Person(String name) {
        this(name, DEFAULT_AGE);
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    //<snipped getName(), setName(), getAge()
}
```

```java
public class Employee extends Person {

    private long employeeId;
    private long departmentId;

    public Employee(String name, int age,
        long departmentId, long employeeId) {

        super(name, age);
        this.departmentId = departmentId;
        this.employeeId = employeeId;
    }
    //<snipped other methods
}
```

We are able to specify the constructor we want to invoke and set attributes for the object.

Demo

Overloading is not just restricted to methods, we are able to apply it to constructors. This is evident within the standard library itself as well!

Constructor Summary

Constructors

Constructor and Description
`Scanner(File source)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(File source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(InputStream source)` Constructs a new Scanner that produces values scanned from the specified input stream
`Scanner(InputStream source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified input stream
`Scanner(Path source)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(Path source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(Readable source)` Constructs a new Scanner that produces values scanned from the specified source.
`Scanner(ReadableByteChannel source)` Constructs a new Scanner that produces values scanned from the specified channel.
`Scanner(ReadableByteChannel source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified channel.
`Scanner(String source)` Constructs a new Scanner that produces values scanned from the specified string.

Constructor Summary, Oracle (https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html)

Overloading is not just restricted to methods, we are able to apply it to constructors. This is evident within the standard library itself as well!

Constructor Summary

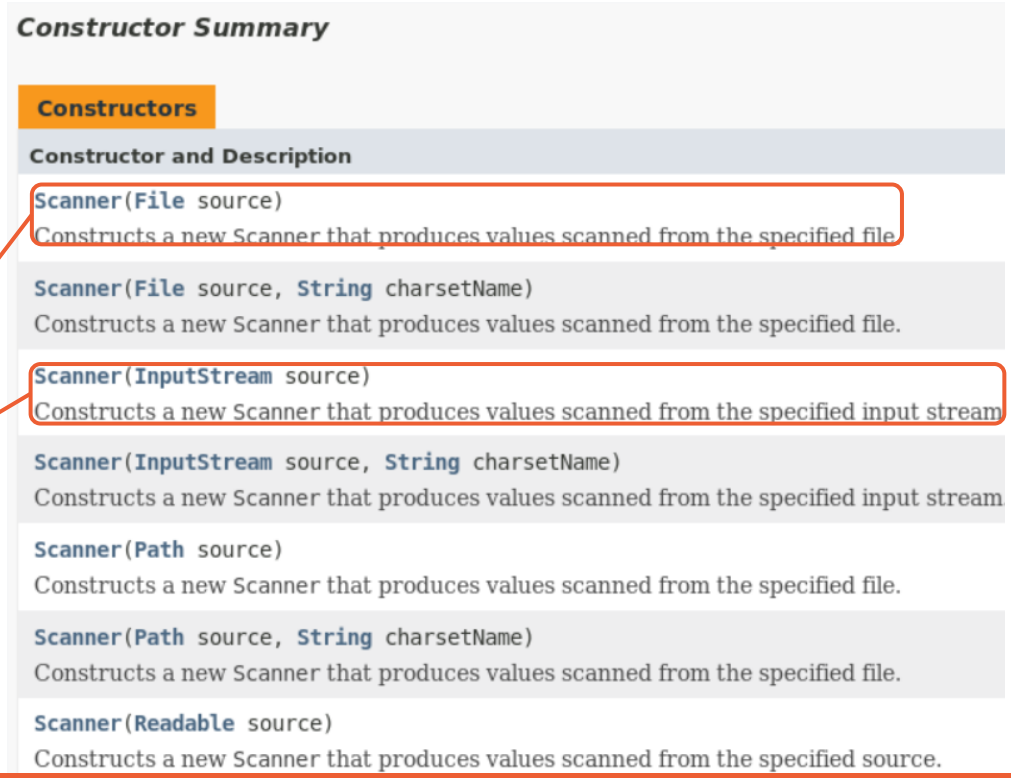
Constructors

Constructor and Description
`Scanner(File source)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(File source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(InputStream source)` Constructs a new Scanner that produces values scanned from the specified input stream.
`Scanner(InputStream source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified input stream.
`Scanner(Path source)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(Path source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(Readable source)` Constructs a new Scanner that produces values scanned from the specified source.
`Scanner(ReadableByteChannel source)` Constructs a new Scanner that produces values scanned from the specified channel.
`Scanner(ReadableByteChannel source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified channel.
`Scanner(String source)` Constructs a new Scanner that produces values scanned from the specified string.

We can see two different methods we have been using for **Files** and the other for **Standard Input**.

Constructor Summary, Oracle (https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html)

Overloading is not just restricted to methods, we are able to apply it to constructors. This is evident within the standard library itself as well!



Constructor Summary

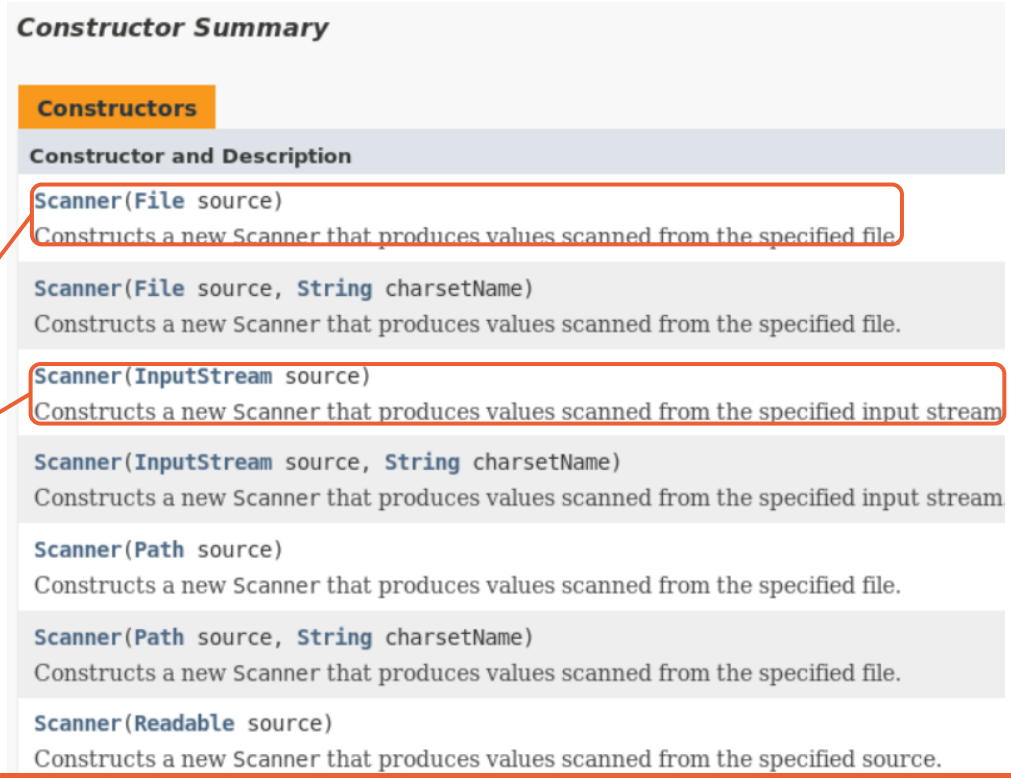
Constructors

Constructor and Description
`Scanner(File source)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(File source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(InputStream source)` Constructs a new Scanner that produces values scanned from the specified input stream.
`Scanner(InputStream source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified input stream.
`Scanner(Path source)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(Path source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(Readable source)` Constructs a new Scanner that produces values scanned from the specified source.

We can see two different methods we have been using for **Files** and the other for **Standard Input**.

Why would the constructor wanting a file as input require handling FileNotFoundException but the InputStream version does not?

Overloading is not just restricted to methods, we are able to apply it to constructors. This is evident within the standard library itself as well!



Constructor Summary

Constructors

Constructor and Description
`Scanner(File source)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(File source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(InputStream source)` Constructs a new Scanner that produces values scanned from the specified input stream.
`Scanner(InputStream source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified input stream.
`Scanner(Path source)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(Path source, String charsetName)` Constructs a new Scanner that produces values scanned from the specified file.
`Scanner(Readable source)` Constructs a new Scanner that produces values scanned from the specified source.

We can see two different methods we have been using for **Files** and the other for **Standard Input**.

Why would the constructor wanting a file as input require handling FileNotFoundException but the InputStream version does not?

Answer: The inputStream is assumed to be opened and controls data being read from it!

IO Classes

We are going to examine how inheritance works within the Java API vai the **java.io** package.

Java has main superclasses are broken into:

- Reader
- Writer
- InputStream
- OutputStream

IO Classes

This can be further generalised that:

- Reader and Writer classes are **Character Stream** classes

- InputStream and OutputStream classes are **Byte Stream** classes.

IO Classes

When inspecting the java.io classes. We can start seeing a pattern in how they have given responsibility to each class.

FileWriter -> Character stream on files

FileOutputStream -> Byte stream on files

Considering IO is not strictly used for file they will utilise similar behaviour from inherited classes for their needs.

BufferedInputStream	A `BufferedInputStream` adds functionality to another input stream-namely, the ability to buffer the input and to support the `mark` and `reset` methods.
BufferedOutputStream	The class implements a buffered output stream.
BufferedReader	Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
BufferedWriter	Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
ByteArrayInputStream	A `ByteArrayInputStream` contains an internal buffer that contains bytes that may be read from the stream.
ByteArrayOutputStream	This class implements an output stream in which the data is written into a byte array.
CharArrayReader	This class implements a character buffer that can be used as a character-input stream.
CharArrayWriter	This class implements a character buffer that can be used as an Writer.
Console	Methods to access the character-based console device, if any, associated with the current Java virtual machine.
DataInputStream	A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.
DataOutputStream	A data output stream lets an application write primitive Java data types to an output stream in a portable way.
File	An abstract representation of file and directory pathnames.
FileDescriptor	Instances of the file descriptor class serve as an opaque handle to the underlying machine-specific structure representing an open file, an open socket, or another source or sink of bytes.
FileInputStream	A `FileInputStream` obtains input bytes from a file in a file system.
FileOutputStream	A file output stream is an output stream for writing data to a `File` or to a `FileDescriptor`.
FilePermission	This class represents access to a file or directory.
FileReader	Convenience class for reading character files.
FileWriter	Convenience class for writing character files.
FilterInputStream	A `FilterInputStream` contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.
FilterOutputStream	This class is the superclass of all classes that filter output streams.
FilterReader	Abstract class for reading filtered character streams.
FilterWriter	Abstract class for writing filtered character streams.
InputStream	This abstract class is the superclass of all classes representing an input stream of bytes.
InputStreamReader	An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified **`charset`**.
LineNumberInputStream	Deprecated *This class incorrectly assumes that bytes adequately represent characters.*
LineNumberReader	A buffered character-input stream that keeps track of line numbers.
ObjectInputStream	An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream.
ObjectInputStream.GetField	Provide access to the persistent fields read from the input stream.
ObjectOutputStream	An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream.
ObjectOutputStream.PutField	Provide programmatic access to the persistent fields to be written to ObjectOutput.
ObjectStreamClass	Serialization's descriptor for classes.
ObjectStreamField	A description of a Serializable field from a Serializable class.
OutputStream	This abstract class is the superclass of all classes representing an output stream of bytes.
OutputStreamWriter	An OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified **`charset`**.

java.io package summary, Oracle (https://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html)

Exceptions

You are bound to have come across an exception while programming with Java. We will be introducing the concept of **throwable** methods.

Looking at the **Scanner** example on **slide 36**. We can observe that some constructors and methods require wrapping around a **try-catch** block.

However, not all exceptions require this.

Like with IO, **Exceptions** have an inheritance hierarchy and contain **specific** error messages to inform the programmer of the error that has occurred.

Exceptions

Whoa! That's a lot of subclasses!

Class Exception

java.lang.Object
    java.lang.Throwable
        java.lang.Exception

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CertificateException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalClassFormatException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, IOException, JAXBException, JMException, KeySelectorException, LambdaConversionException, LastOwnerException, LineUnavailableException, MarshalException, MidiUnavailableException, MimeTypeParseException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NotBoundException, NotOwnerException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, ReflectiveOperationException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ScriptException, ServerNotActiveException, SOAPException, SQLException, TimeoutException, TooManyListenersException, TransformerException, TransformException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, UserException, XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException

Exception Class, Oracle (https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/Exception.html)

Exceptions

Let's tackle the two major **Exception** classes that dictate how the rest operate.

- Exception and any subclasses (with the exception of **RuntimeException**) is a **checked exception**

  This means that when a method **can** throw the exception, the programmer **must** handle it using a **try-catch** block.

- **RuntimeException** and any subclasses, is an **unchecked exception.** The programmer does not need to handle this case but can catch if they want to.

  This should be a case where the program should crash.

Checked Exception

Let’s examine the following

```
public void imGonnaCrash() throws Exception {
    throw new Exception("Definitely crashing!");
}
```

Checked Exception

Let’s examine the following

```
public void imGonnaCrash() throws Exception {
    throw new Exception("Definitely crashing!");
}
```

Since the method can throw a **checked exception** we are required it to handle it when we call it.,

Checked Exception

Let's examine the following

```
public void imGonnaCrash() throws Exception {
    throw new Exception("Definitely crashing!");
}
```

Where we throw the exception, typically this is in some kind of if statement.

Since the method can throw a **checked exception** we are required it to handle it when we call it.

Checked Exception

Let’s examine the following

```
public void imGonnaCrash() throws Exception {
    throw new Exception("Definitely crashing!");
}
```

Within our main method we **cannot proceed** with the following.

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

Checked Exception

Let’s examine the following

```
public void imGonnaCrash() throws Exception {
    throw new Exception("Definitely crashing!");
}
```

We are **forced** to catch it by the compiler;.

```
public static void main(String[] args) {
    try {
        imGonnaCrash();
    } catch(Exception e) {
       e.printStackTrace();
    }
}
```

Runtime Exception

Let's examine the following

```
public void imGonnaCrash() {
    throw new RuntimeException("Definitely crashing!");
}
```

Where the compiler will **not** force the programmer to handle a **RuntimeException.**

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

Let’s see how throwing exceptions work

See you next time!