INFO1113 Object-Oriented Programming

Week 3B: Methods and IO
Static and non-static context and Text IO

Slides developed by Tyson Thomas

Copyright Warning

Topics

- **`this`** keyword and non-static context
- Mixing static and non-static context
- Input and Output
- Text I/O

`this` keyword

We'll expand on the **this** keyword and how it can help with eliminating ambiguity but also used for passing an object reference within an instance context.

The **`this`** keyword allows the programmer to refer to the object while within an **instance** method context. We cannot use the keyword within a **static** context.

It is also used for referring to another constructor to allow for code reusability. (We will elaborate on this in Week 5!)

Refer to Chapter 5.1, pages 322-323, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

`this` keyword

Let's say we have this issue:

```
public class Postcard {

    String sender;
    String receiver;
    String address;
    String contents;

    public Postcard(String sender, String receiver, String address,
        String contents) {

        //Blasts! Foiled by ambiguity!
    }
}
```

Refer to Chapter 5.1, pages 322-323, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

this keyword

Let's say we have this issue:

```
public class Postcard {

    String sender;
    String receiver;
    String address;
    String contents;

    public Postcard(String sender, String receiver, String address,
        String contents) {

        //Blasts! Foiled by ambiguity!
    }
}
```

We can't specify **sender = sender;** because the compiler cannot determine what is inferred by the statement.

Refer to Chapter 5.1, pages 322-323, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

this keyword

Obvious solution

```
public class Postcard {

    String sender;
    String receiver;
    String address;
    String contents;

    public Postcard(String s, String r, String a, String c) {
        sender = s;
        receiver = r;
        address = a;
        contents = c;
    }
}
```

Refer to Chapter 5.1, pages 322-323, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

this keyword

Obvious solution

```
public class Postcard {

    String sender;
    String receiver;
    String address;
    String contents;

    public Postcard(String s, String r, String a, String c) {
        sender = s;
        receiver = r;
        address = a;
        contents = c;
    }
}
```

Cool! We have now exchanged **readability** for **cryptic** letters. Fair exchange? This will compile but we will not be able to generate documentation easily from this.

Refer to Chapter 5.1, pages 322-323, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Can we eliminate ambiguity and also have readability?

this keyword

Yes!

```
public class Postcard {

    String sender;
    String receiver;
    String address;
    String contents;

    public Postcard(String sender, String receiver, String address,
        String contents) {

        this.sender = sender;
        this.receiver = receiver;
        this.address = address;
        this.contents = contents;
    }
}
```

We have used the **this** keyword to eliminate ambiguity within this block of code.

this corresponds to the **instance** within the block.

Refer to Chapter 5.1, pages 322-323, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

this keyword

Yes!

```
public class Postcard {

    String sender;
    String receiver;
    String address;
    String contents;

    public Postcard(String sender, String receiver, String address,
        String contents) {

        this.sender = sender;
        this.receiver = receiver;
        this.address = address;
        this.contents = contents;
    }
}
```

This seems **very familiar**! Oh yeah, it's like the **self** variable in **python.**

We have used the **this** keyword to eliminate ambiguity within this block of code.

this corresponds to the **instance** within the block.

Refer to Chapter 5.1, pages 322-323, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

this keyword

Yes!

```
public class Postcard {

    String sender;
    String receiver;
    String address;
    String contents;

    public Postcard(String sender, String receiver, String address,
        String contents) {

        this.sender = sender;
        this.receiver = receiver;
        this.address = address;
        this.contents = contents;
        System.out.println(this)

    }
}
```

```
Postcard p1 = new PostCard(...);
Postcard p2 = new PostCard(...);
System.out.println(p1);
System.out.println(p2);
```

What would happen if we tried to output this out?

Refer to Chapter 5.1, pages 322-323, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Let’s see what happens

Instance Methods

We covered instance methods in the previous lecture but now let's expand on them and discuss about **static** and **instance** contexts.

We will be revisiting the **this** keyword again in this section to help understand how it is applied.

Within the context of an instance method, it refers to the current calling object. It cannot be used within a static method as it is unable to refer to the calling object.

Refer to Chapter 6.2, pages 433-440, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Instance Method Reinterpreted

```
public class Postcard {

    String sender;
    String receiver;
    <...snip...>

    public void setSender(String sender) {
        this.sender = sender;
    }

}
```

Instance Method Reinterpreted

```
public class Postcard {

    String sender;
    String receiver;
    <...snip...>

    public void setSender(String sender) {
        this.sender = sender;
    }

}
```

this is not just limited to the constructor we are able to use it within **instance methods**.

However! How could we reinterpret an instance method? How is the object given to the method?

```
Postcard p1 = new PostCard(...);
p1.setSender("Bob");
```

Instance Method Reinterpreted

```
public class Postcard {

    String sender;
    String receiver;
    <...snip...>

    public static void setSender(Postcard p, String sender) {
        p.sender = sender;
    }

}
```

```
Postcard p1 = new PostCard(...);
Postcard.setSender(p1, “Masa”);
```

One may consider it **magic** where the method knows the object without it being passed to it.

Although you would never write something like this for the purpose of creating a setter or getting it completes how the object is passed and how the method is expanded.

Instance and static methods

Let's examine the following code segment.

```
public class Postcard {

    String sender;
    String receiver;
    boolean received;
    <...snip...>

    public static boolean inTransit() {
        return  !received;
    }

    public void setSender(String sender) {
        this.sender = sender;
    }

}
```

Instance and static methods

Let's examine the following code segment.

```
public class Postcard {

    String sender;
    String receiver;
    boolean received;
    <...snip...>

    public static boolean inTransit() {
        return  !received;
    }

    public void setSender(String sender) {
        this.sender = sender;
    }

}
```

This **static** method is attempting to utilise an **instance** variable. Why is this a problem?

Instance and static methods

Let's examine the following code segment.

```
public class Postcard {

    String sender;
    String receiver;
    boolean received;
    <...snip...>

    public static boolean inTransit() {
        return  !received;
    }

    public void setSender(String sender) {
        this.sender = sender;
    }

}
```

This **static** method is attempting to utilise an **instance** variable. Why is this a problem?

Because it isn't referring to an object Instance methods are not allowed in this context.

Instance and static methods

Let's examine the following code segment.

```
public class Postcard {

    String sender;
    String receiver;
    boolean received;
    <...snip...>

    public boolean inTransit() {
        return !received;
    }

    <...snip...>
    public static boolean hasArrived(Postcard p) {
        if(!p.inTransit()) { return true; }
        else { return false; }
    }
}
```

Instance and static methods

Let's examine the following code segment.

```
public class Postcard {

    String sender;
    String receiver;
    boolean received;
    <...snip...>

    public boolean inTransit() {
        return !received;
    }

    <...snip...>
    public static boolean hasArrived(Postcard p) {
        if(!p.inTransit()) { return true; }
        else { return false; }
    }
}
```

This **static** method is attempting to utilise an **instance** method attached to an object. Is there an issue?

Instance and static methods

Let's examine the following code segment.

```
public class Postcard {

    String sender;
    String receiver;
    boolean received;
    <...snip...>

    public boolean inTransit() {
        return !received;
    }

    <...snip...>
    public static boolean hasArrived(Postcard p) {
        if(!p.inTransit()) { return true; }
        else { return false; }
    }
}
```

This **static** method is attempting to utilise an **instance** method attached to an object. Is there an issue?

Nope! Simply, there is an object instantiated and we are able to utilise method.

Let’s get tricky

Let’s examine the following code segment.

```
public class Postcard {

    String sender;
    String receiver;
    boolean received;
    <...snip...>

    public boolean alreadyArrived() {
        return hasArrived(this);
    }

    <...snip...>
    public static boolean hasArrived(Postcard p) {
        if(!p.inTransit()) { return true; }
        else { return false; }
    }
}
```

Let's get tricky

Let's examine the following code segment.

```
public class Postcard {

    String sender;
    String receiver;
    boolean received;
    <...snip...>

    public boolean alreadyArrived() {
        return hasArrived(this);
    }

    <...snip...>
    public static boolean hasArrived(Postcard p) {
        if(!p.inTransit()) { return true; }
        else { return false; }
    }
}
```

We have an **instance** method invoking a **static method** while also using the **this** keyword.

Is this correct?

Let's get tricky

Let's examine the following code segment.

```
public class Postcard {

    String sender;
    String receiver;
    boolean received;
    <...snip...>

    public boolean alreadyArrived() {
        return hasArrived(this);
    }

    <...snip...>
    public static boolean hasArrived(Postcard p) {
        if(!p.inTransit()) { return true; }
        else { return false; }
    }
}
```

We have an **instance** method invoking a **static method** while also using the **this** keyword.

Yes! We are just passing the instance to a static function. This is no different from what we have done before but we are using the **this** keyword.

So let’s construct this

Input and Output

We are able to read and write to devices. Specifically we will be focusing on reading and writing to storage.

If we are intending to use data stored in a file, we have to understand how that data is stored and what will be an appropriate tool for the job.

What kind of data is stored in the following files?

- HelloWorld.java
- Cat.jpg
- Program.exe
- TODO.txt

Refer to Chapter 10.1, pages 776-777, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Files

I/O Classes.

Within the java api we have access to a large range of I/O classes.

You have already been using the **Scanner** class for reading content from **standard input**. However we are able to interact with a variety of sources.

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Using scanner

We can now use Scanner to read files. As the name implies it **Scan's** for input and provides functionality to read it.

```
import java.io.File;
import java.util.Scanner;
public class FileHandle {

    public static void main(String[] args) {
        File f = new File("README");
        Scanner scan = new Scanner(f);

    }
}
```

We have **File** object that will abstract represent the file stored at a **Path**.

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Using scanner

We can now use Scanner to read files. As the name implies it **Scan's** for input and provides functionality to read it.

```
import java.io.File;
import java.util.Scanner;
public class FileHandle {

    public static void main(String[] args) {
        File f = new File("README");
        Scanner scan = new Scanner(f);

    }
}
```

We have **File** object that will abstract represent the file stored at a **Path**.

Scanner accepts a file as an argument and is able read contents there.

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Using scanner

We can now use Scanner to read files. As the name implies it **Scan's** for input and provides functionality to read it.

```
import java.io.File;
import java.util.Scanner;
public class FileHandle {

    public static void main(String[] args) {
        File f = new File("README");
        Scanner scan = new Scanner(f);

    }
}
```

We have **File** object that will abstract represent the file stored at a **Path**.

Scanner accepts a file as an argument and is able read contents there.

Unfortunately…. This code won't compile **:(** Why would this be the case?

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Compiler at it again!

```
> javac FileHandle.java

FileHandle.java:7: error: unreported exception FileNotFoundException;
Must be caught or declared to be thrown
        Scanner scan = new Scanner(f);

1 error
```

As with most **IO operations** we will be required to perform some exception handling.

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Using scanner

We can now use Scanner to read files. As the name implies it **Scan's** for input and provides functionality to read it.

```
import java.io.File;
import java.util.Scanner;
public class FileHandle {

    public static void main(String[] args) {
        File f = new File("README");
        try {
            Scanner scan = new Scanner(f);
        } catch (FileNotFoundException e) {
            System.err.println("File not found!");
        }
    }
}
```

If the file does not exist we are unable to read from it. This allows the programmer to have a branch for both. A **state** where we **can read data** and one **without reading data**.

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Using scanner

We can now use Scanner to read files. As the name implies it **Scan's** for input and provides functionality to read it.

```
import java.io.File;
import java.util.Scanner;
public class FileHandle {

    public static void main(String[] args) {
        File f = new File("README");
        try {
            Scanner scan = new Scanner(f);
        } catch (FileNotFoundException e) {
            System.err.println("File not found!");
        }
    }
}
```

Java forces us to provide some checks to ensure we are handling certain except cases correctly.

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

How is reading performed?

Reading any kind of file is analogous to working with ***contiguous memory***.

Let's say we have the following file called **"today.txt"** which contains the following contents:

```
Today is great!
```

This can be represented with the following array:

T	o	d	a	y		i	s		g	r	e	a	t	!	\0

```
File f = new File("README");
Scanner scan = new Scanner(f);
scan.next(); //Today
scan.next(); //is
scan.next(); //great!
```

Scanner itself doesn't ***support*** reading **character by character**. Reasoning behind this is because the idea of a character depends on how it is encoded

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

How is reading performed?

Reading any kind of file is analogous to working with ***contiguous memory***.

Let's say we have the following file called **"today.txt"** which contains the following contents:

```
Today is great!
```

This can be represented with the following array:

T	o	d	a	y		i	s		g	r	e	a	t	!	\0

```
File f = new File("README");
Scanner scan = new Scanner(f);
scan.next(); //Today
scan.next(); //is
scan.next(); //great!
```

Executing the following line will move the cursor to the next space (or whatever token we want to separate words by).

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

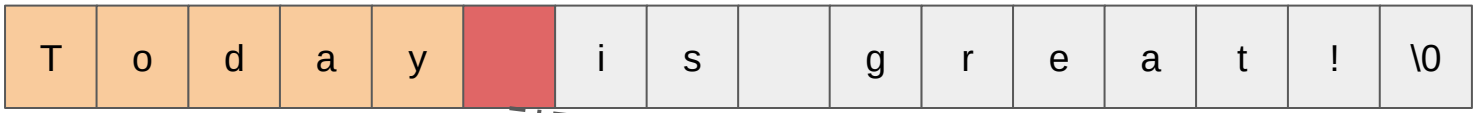
How is reading performed?

Reading any kind of file is analogous to working with ***contiguous memory***.

Let's say we have the following file called **"today.txt"** which contains the following contents:

```
Today is great!
```

This can be represented with the following array:




```
File f = new File("README");
Scanner scan = new Scanner(f);
scan.next(); //Today
scan.next(); //is
scan.next(); //great!
```

The cursor has moved once next() has been called.

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

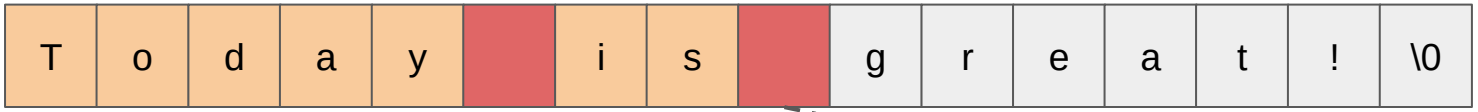
How is reading performed?

Reading any kind of file is analogous to working with ***contiguous memory***.

Let's say we have the following file called **"today.txt"** which contains the following contents:

```
Today is great!
```

This can be represented with the following array:




```
File f = new File("README");
Scanner scan = new Scanner(f);
scan.next(); //Today
scan.next(); //is
scan.next(); //great!⬅
```

The cursor has moved once next() has been called.

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

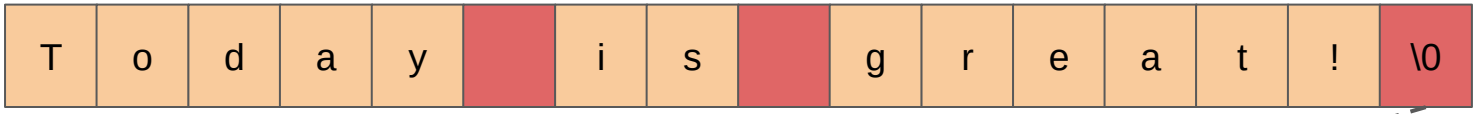
How is reading performed?

Reading any kind of file is analogous to working with ***contiguous memory***.

Let's say we have the following file called **"today.txt"** which contains the following contents:

```
Today is great!
```

This can be represented with the following array:



The cursor has moved once next() has been called.

```
File f = new File("README");
Scanner scan = new Scanner(f);
scan.next(); //Today
scan.next(); //is
scan.next(); //great!
```

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Writing text data

As discussed prior, Scanner only performs reading an object. So how about writing?

PrintWriter allows for printing formatted representations of objects to a text-output stream.

```
import java.io.File;
import java.io.PrintWriter;
import java.io.FileNotFoundException;
public class FileHandle {
    public static void main(String[] args) {
        File f = new File(“README”);
        try {
            PrintWriter writer = new PrintWriter(f);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Writing text data

As discussed prior, Scanner only performs reading an object. So how about writing?

PrintWriter allows for printing formatted representations of objects to a text-output stream.

```
import java.io.File;
import java.io.PrintWriter;
import java.io.FileNotFoundException;
public class FileHandle {
    public static void main(String[] args) {
        File f = new File("README");
        try {
            PrintWriter writer = new PrintWriter(f);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

We have a class that allows for writing of formatted data.

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Writing text data

```
import java.io.File;
import java.io.PrintWriter;
import java.io.FileNotFoundException;
public class FileHandle {
    public static void main(String[] args) {
        File f = new File("README");
        try {
            PrintWriter writer = new PrintWriter(f);
            writer.println(1.0);
            writer.println(120);
            writer.println("My String!");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Writing text data

```
import java.io.File;
import java.io.PrintWriter;
import java.io.FileNotFoundException;
public class FileHandle {
    public static void main(String[] args) {
        File f = new File("README");
        try {
            PrintWriter writer = new PrintWriter(f);
            writer.println(1.0);
            writer.println(120);
            writer.println("My String!");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

We have a class that allows for writing of formatted data. It's methods are very similar to that of **System.out.** That is no coincidence!

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Writing text data

```
import java.io.File;
import java.io.PrintWriter;
import java.io.FileNotFoundException;
public class FileHandle {
    public static void main(String[] args) {
        File f = new File("README");
        try {
            PrintWriter writer = new PrintWriter(f);
            writer.println(1.0);
            writer.println(120);
            writer.println("My String!");

            writer.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
```

This will write output 1.0, 120 and "My String!" to the file **README**.

We have a class that allows for writing of formatted data. It's methods are very similar to that of **System.out.** That is no coincidence!

Refer to Chapter 10.1, pages 778-785, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

See you next time!