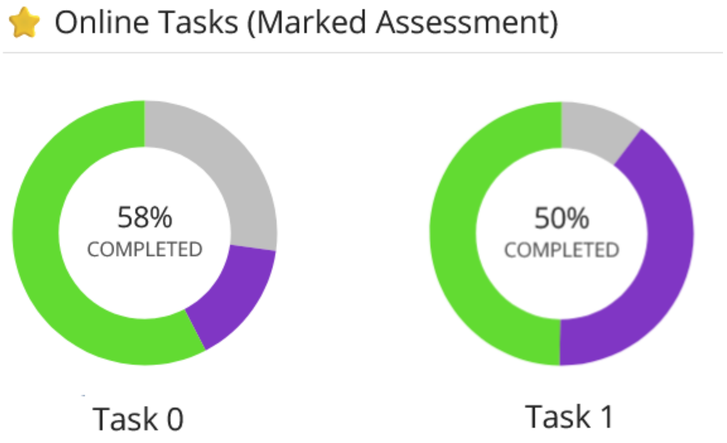
INFO1113 / COMP9003
Object-Oriented Programming

Lecture 3

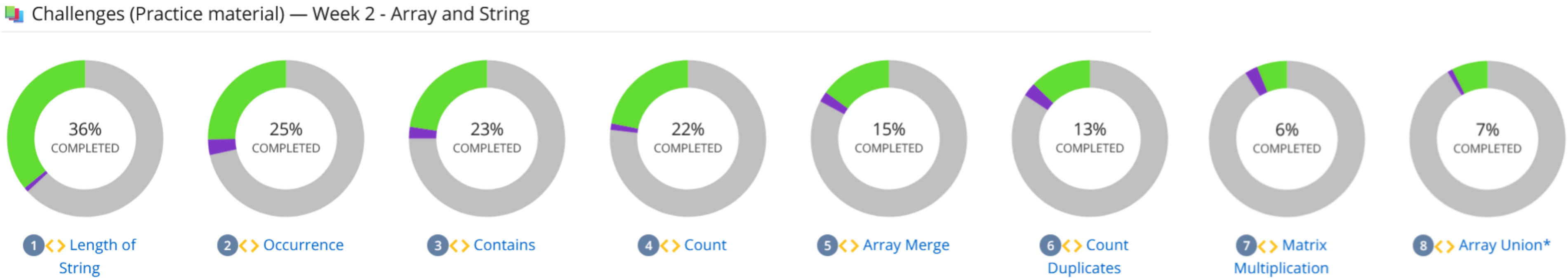
THE UNIVERSITY OF SYDNEY

Contents developed by Tyson Thomas

Always try to be in the green zone!



Online Tasks (Marked Assessment)



Challenges (Practice material) — Week 2 - Array and String

Early Feedback Task

- Runs during week3 tutorial
- Restricted open book
    - You may use the Javadoc, lecture slides, and tutorial resources, but *not* any LLMs like ChatGPT.
- Covers content from Weeks 1 and 2
- It will be an online coding task on Ed during your tutorial
- You are allowed to use your laptop
- Everyone starts at the same time
    - Duration: 40 minutes
    - Start Time: As determined by your tutor

Acknowledgement of Country

I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.

I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.

Copyright Warning

Topics: Part A

- Classes
- Object attributes
- Instance Methods
- UML Class Diagram

Classes
“Where Reference Types Come From”

Classes

There is a clear distinction between a **primitive type** and a **reference type** but how is the distinction made?

Reference Types are **Classes**.

We have already used classes within our programs since the start of semester. However, now we are able to define our own classes.

Most programming languages have some mechanism of structuring data for reuse. In Java, Class is a primary way of structuring data.

Refer to Chapter 5.3, pages 362-366, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

But what are they?

What’s a class

“A class defines a type or kind of object. It is a blueprint for defining the objects. All objects of the same class have the same kinds of data and the same behaviours. When the program is run, each object can act alone or interact with other objects to accomplish the program’s purpose.”

Sometimes it is simply conveyed as a **blueprint/template/concept** of an object.

Refer to Chapter 5.1, page 303, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

What's a class

Every java program we have ever written so far has included the idea of a class in some form.

However we have never **instantiated** an instance of our own class yet.

We have been merely using inbuilt classes within java such as:

- Scanner
- String
- StringBuilder

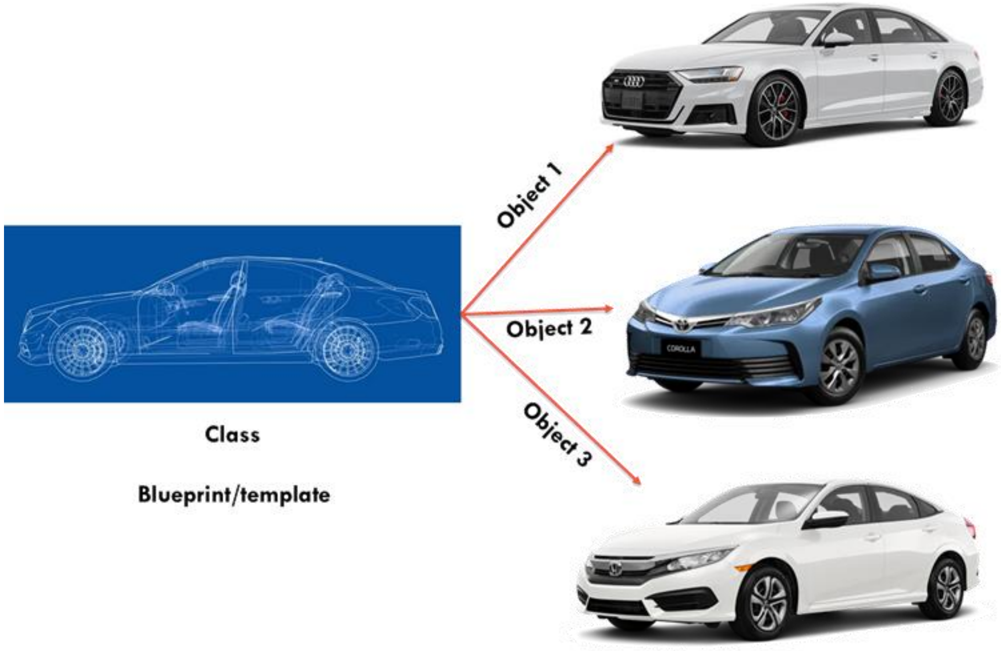
Objects

Shiny **new** keyword! As discussed before, this allocates memory and instantiates an object.

We have to ask where this **method** exists

Objects are *instance* of a particular *class*.

```
Car toyota = new Car(“fuel”, “white”);
Car mazda = new Car(“Hybrid”, “blue”);
Car tesla = new Car(“electric”, “white”);
```



Can I make my own class?

Classes

Yes!

However let’s start off with a basic **class definition**.

```
public class Cupcake {
  public boolean delicious;
  public String name;
}
```

This is the **body** of the class. We define **attributes** of **object** within this space.

We can **instantiate** this class with the following line of code

```
  Cupcake c = new Cupcake();
```

Refer to Chapter 5.3, pages 306-307, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Classes

Yes!

However let’s start off with a basic **class definition**.

```
public class Cupcake {
  public boolean delicious;
  public String name;
}
```

This is the **body** of the class. We define **attributes** of **object** within this space.

We can **instantiate** this class with the following line of code

```
Cupcake c = new Cupcake();
```

Declares a **Cupcake** object.

Java is **allocating** space for a **Cupcake** object and **invoking the constructor** to initialise it.

Constructors

Every class in **Java** has a **Constructor** even if it is not **explicitly defined**.

Extending our **Cupcake** class we can write our own constructor.

```
public class Cupcake {
   public boolean delicious;
   public String name;

   public Cupcake() { /* NO OP */ }
}
```

Refer to Chapter 6.1, pages 419 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Constructors

Every class in **Java** has a **Constructor** even if it is not **explicitly defined**.

Extending our **Cupcake** class we can write our own constructor.

```
public class Cupcake {
   public boolean delicious;
   public String name;

   public Cupcake() { /* NO OP */ }
}
```

This looks like a method but has no **return type?**
The **constructor's** role is to **construct** an **object** of the **type Cupcake.**

Constructors

Every class in **Java** has a **Constructor** even if it is not **explicitly defined**.

Extending our **Cupcake** class we can write our own constructor.

```
public class Cupcake {
   public boolean delicious;
   public String name;

   public Cupcake() {
      delicious = true;
      name = "Chocolate Cupcake";
   }
}
```

We can expand on this to provide **default values**

Constructors

Every class in **Java** has a **Constructor** even if it is not **explicitly defined**.

Extending our **Cupcake** class we can write our own constructor.

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake(boolean isTasty) {
        delicious = isTasty;
        name = "Chocolate Cupcake";
    }
}
```

We can expand on this to provide **default values** and **parameters** for our constructor . We can then invoke the parameter with arguments that relate to the object.

Let’s make some classes!

Objects

Now using our nice cupcake class, let’s see what we can do with it!

```
public class Cupcake {
    public boolean delicious;
    public String name;

    public Cupcake(boolean isTasty) {
        delicious = isTasty;
        name = "Chocolate Cupcake";
    }
}
```

We can instantiate our own instance!

```
Cupcake mine = new Cupcake(true);
Cupcake toShare = new Cupcake(false);
System.out.println(mine.delicious);
System.out.println(toShare.delicious);
```

Refer to Chapter 6.1, pages 424-429 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Objects

Now using our nice cupcake class, let’s see what we can do with it!

```
public class Cupcake {
   public boolean delicious;
   public String name;

   public Cupcake(boolean isTasty) {
      delicious = isTasty;
      name = "Chocolate Cupcake";
   }
}
```

We can instantiate our own instance!

```
Cupcake mine = new Cupcake(true);
Cupcake toShare = new Cupcake(false);
System.out.println(mine.delicious);
System.out.println(toShare.delicious);
```

We have instantiated a cupcake to the variable **mine** and inputted **true**. This will set the **delicious** attribute to **true.**

I have deliberately provided a bland cupcake to everyone by setting the **isTasty** to **false.**

Refer to Chapter 6.1, pages 424-429 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Objects

Now using our nice cupcake class, let’s see what we can do with it!

```
public class Cupcake {
   public boolean delicious;
   public String name;

   public Cupcake(boolean isTasty) {
      delicious = isTasty;
      name = "Chocolate Cupcake";
   }
}
```

We can instantiate our own instance!

```
Cupcake mine = new Cupcake(true);
Cupcake toShare = new Cupcake(false);
System.out.println(mine.delicious);
System.out.println(toShare.delicious);
```

We can access the attributes of the object by using the **.**<attribute>

You may have already of picked up on that from using **Scanner** and **String**

Refer to Chapter 6.1, pages 424-429 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Let’s get back and use that class!

Instance Methods

We're finally getting rid of the static (training wheels off!)

Syntax:
[final] return_type name ([parameters])

An instance method operates on attributes associated with the instance. These methods can **only** be used with an object.

Instance Methods

Let’s extend our **Cupcake** class!

```
public class Cupcake {
   public boolean delicious;
   private String name;

   public Cupcake(boolean isTasty, String cupcakeName) {
      delicious = isTasty;
      name = cupcakeName;
   }

   public void setName(String n) { name = n; }

   public String getName() { return name; }
}
```

Refer to Chapter 5.2, pages 342-344 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Instance Methods

Let’s extend our **Cupcake** class!

```
public class Cupcake {
    public boolean delicious;
    private String name;

    public Cupcake(boolean isTasty, String cupcakeName) {
        delicious = isTasty;
        name = cupcakeName;
    }

    public void setName(String n) { name = n; }

    public String getName() { return name; }
}
```

private modifier limits how and where the attribute can be accessed. **public** allows access outside of the class while **private** limits itself to the scope of the class.

A **setter** method specified. This allows us to modify the **name** attribute.

A **getter** method has been specified here. This method merely returns the attribute **name.**

Instance Methods

Let’s extend our **Cupcake** class!

```
public class Cupcake {
   public boolean delicious;
   private String name;

   public Cupcake(boolean isTasty, String cupcakeName) {
      delicious = isTasty;
      name = cupcakeName;
   }

   public void setName(String n) { name = n; }

   public String getName() { return name; }
}
```

```
Cupcake mine = new Cupcake(true, ”My Cupcake!”);
Cupcake toShare = new Cupcake(false, ”Everyone’s Cupcake”);
mine.setName("My Cupcake, Don’t touch!");
```

Instance Methods

We want to eat the cupcake

```
public class Cupcake {
   public boolean delicious;
   private String name;
   private boolean eaten;

   public Cupcake(boolean isTasty, String cupcakeName) {
      delicious = isTasty;
      name = cupcakeName;
      eaten = false;
   }

   public void setName(String n) { name = n; }

   public String getName() { return name; }

   public void eat() { eaten = true; }

}
```

Instance Methods

We want to eat the cupcake

```
public class Cupcake {
   public boolean delicious;
   private String name;
   private boolean eaten;

   public Cupcake(boolean isTasty, String cupcakeName) {
      delicious = isTasty;
      name = cupcakeName;
      eaten = false;
   }

   public void setName(String n) { name = n; }

   public String getName() { return name; }

   public void eat() { eaten = true; }

}
```

Now we have an extra property called **eaten** and we can write a method called **eat()** that will change the state of the object.

Instance Methods

```
public class Cupcake {
  public boolean delicious;
  private String name;
  private boolean eaten;

  public Cupcake(boolean isTasty, String cupcakeName) {
    delicious = isTasty;
    name = cupcakeName;
    eaten = false;
  }

  public void setName(String n) { name = n; }

  public String getName() { return name; }

  public void eat() {
    if(!eaten) {
      System.out.println(“That was nice!”);
    eaten = true;
  }
}
```

Expanding on this method, we can output to the user when it has been eaten.

Let’s extend this class and test it!

UML

Unified Modelling Language, a visual language to assist with designing applications and systems.

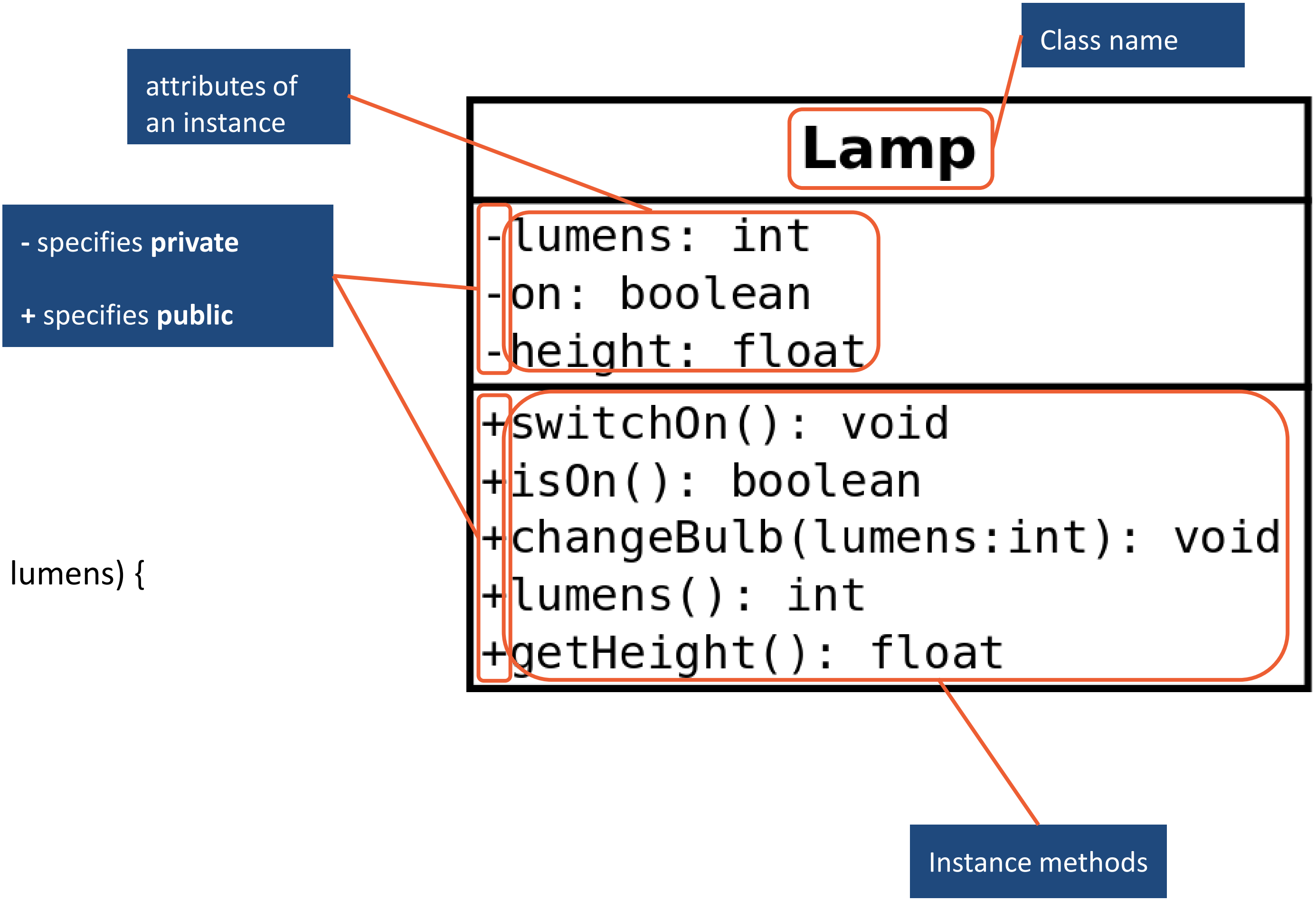
Specifically in this course we are focused on **UML Class Diagrams**.

Class diagrams allow us to design classes prior to implementing them. Giving the ability to model the system without implementing it first.

UML Class Diagram

```
public class Lamp {
   private int lumens;
   private boolean on;
   private float height;

   public void switchOn() {
      ……
   }
   public boolean isOn() {
      ……
   }
   public void changeBulb(int lumens) {
      ……
   }
   public int lumens() {
      ……
   }
   public float getHeight() {
      ……
   }
}
```



Class name

attributes of an instance

- specifies **private**

+ specifies **public**

Instance methods

```
Lamp
-lumens: int
-on: boolean
-height: float
+switchOn(): void
+isOn(): boolean
+changeBulb(lumens:int): void
+lumens(): int
+getHeight(): float
```

Okay but what about the + and - annotations?

Refer to Chapter 5.3, page 374 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Let’s take a break!

Topics: Part B

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

this **keyword**

We’ll expand on the **this** keyword and how it can help with eliminating ambiguity and 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) {
            sender = sender;
            ………
    }
}
```

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) {
            sender = sender;          //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;
    }
}
```

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.

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;
    }
}
```

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.

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;
    }
}
```

```
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?

Let’s see what happens

Instance Methods

We covered instance methods before 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, “Adam”);
```

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; }
       }
}
```

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

```
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

```
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

```
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?

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

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.txt”);
        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 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

```
import java.io.File;
import java.util.Scanner;
import java.io.FileNotFoundException;
public class FileHandle {
    public static void main(String[] args) {
        File f = new File(“README.txt”);
        try {
            Scanner scan = new Scanner(f);
        } catch (FileNotFoundException e) {
            System.out.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**.

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 **"README.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.txt");
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 **“README.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.txt”);
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)

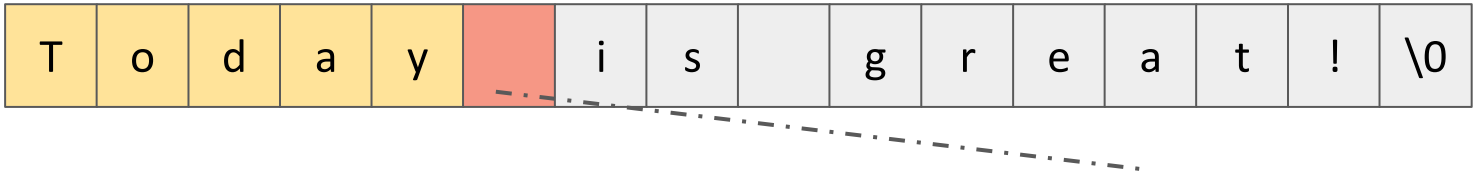
How is reading performed?

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

Let’s say we have the following file called **“README.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.txt”);
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)

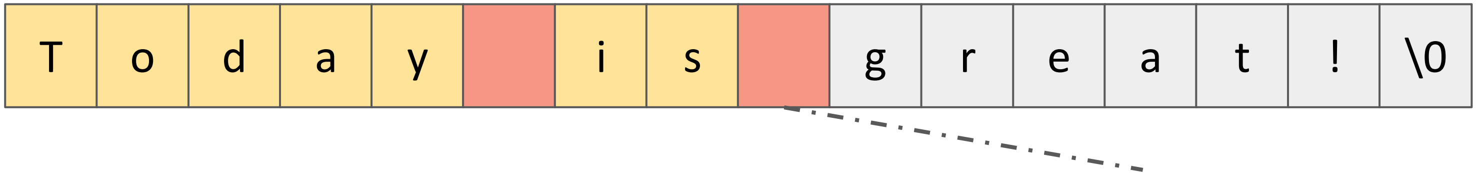
How is reading performed?

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

Let’s say we have the following file called **“README.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.txt”);
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)

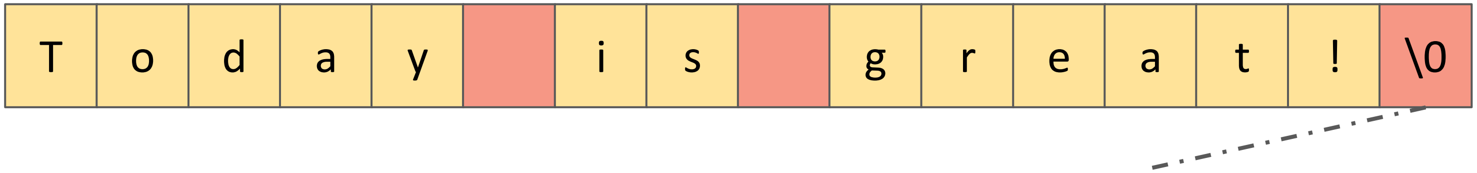
How is reading performed?

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

Let’s say we have the following file called **“README.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.txt”);
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)

Demonstration: Reading from a file

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.txt”);
        try {
            PrintWriter writer = new PrintWriter(f);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

We have a class that allows writing 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.txt”);
        try {
            PrintWriter writer = new PrintWriter(f);
            writer.println(1.0);
            writer.println(120);
            writer.println(“My String!”);
            writer.close();
        } 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!

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

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

Demonstration: Writing in a file

See you next time!

THE UNIVERSITY OF SYDNEY