

The Ubiquitous Object class

CPSC 2150

The Ubiquitous Class: `Object`

- *Every* class in Java extends (inherits from) `Object`, which is a special built-in class that provides default implementations for the following instance methods (among a few others that are not so important):

```
boolean equals(Object obj)
```

```
String toString()
```

Recall: Inheritance

- If object A inherits from (or `extends`) object B
 - Object A will have all of the data fields and methods of object B
 - This is done automatically, no extra coding needed
- Since *every* class in Java automatically extends `Object`, every class automatically has the methods defined in `Object`
 - However, since `Object` has no data fields, these methods don't really do anything
 - It really just ensures that every class has the same name for these important methods
 - **Not:**
 - `A.equals()`
 - `B.isEqual()`
 - `C.checkEqual()`
 - `etc.`

equals

boolean equals(Object obj)

- Reports whether **this** is equal to obj.

- Ensures:

equals = (**this** == obj)

`equals`

`boolean`

- Rep
- Ensures:

```
equals = (this == obj)
```

The default implementation in `Object` checks reference equality, though we expect object value equality! So, we (almost always) need to override this method.

toString

`String toString()`

- Returns the string representation of **this**.

- Ensures:

*toString = [the string
representation of **this**]*

toString

`String toString()`

- Returns the string representation of **this**.
- Ensures:

*toString = [the string
representation of **this**]*

The default implementation in `Object` returns a `String` that shows the reference value of **this**, though we expect it to show the object value! So, we (almost always) need to override this method.

“@Override” Annotation

- When writing the code for the body of a method
 - that overrides a method in a class being *extended*you preface the method body with an ***@Override annotation***

Example “@Override”

```
@Override  
public String toString() {  
    ...  
}
```

Overriding `equals` and `toString`

- You always want to override the `equals` operator and the `toString` operator
- Even if you don't intend to use them, someone else might
- Even if no one writes code to call them, it already exists
- `contains()` method in `List`
 - Calls `equals` in that code
- Since every class inherits from the `Object` class, we can always assume there is *some* definition for those methods
 - They just may not do what we want them to do, so we override them in our classes

Overriding equals

- There is an issue with overriding the `equals` method
- `Object.equals(Object obj)`
 - `equals` expects an object of type `Object`
 - But we want this to work with our user defined class
- Since every class extends `Object`, we can pass in any class to the `equals` operation without any compiler issues
 - Every class is also an `Object`
- However, we can't call any methods that belong to the specific class and not to object
 - At least not yet
- Also, someone could call `A.equals(B)` when `A` is a `Pencil` and `B` is a `Car`!

instanceOf

- There is a special operator in Java called `instanceOf`
- **Syntax:** `<object> instanceof <Class>`
- Returns *true* if the object is of type class
 - *False* otherwise

instanceOf

```
public class Pencil {  
    private String color;  
    private int length;  
    ...  
    @Override  
    public boolean equals(Object obj) {  
        if (!(obj instanceof Pencil))  
            return false;  
        ...  
    }  
}
```

Casting

- Now we know that `obj` is the right type, but if we start calling methods, we'll get a syntax error
 - `obj` has a data type of `Object`
 - `Object` does not have member called `length` or `color`
- We need to cast
 - The same way we did with primitive types
- Remember, casting is us promising that it will work, so make sure you used `instanceOf` to check the type first

```
public class Pencil {  
    private String color;  
    private int length;  
    ...  
    @Override  
    public boolean equals(Object obj) {  
        if (!(obj instanceof Pencil))  
            return false;  
        Pencil p = (Pencil) obj;  
        if (this.color.equals(p.color)  
            && this.length == p.length)  
        {  
            return true;  
        }  
        return false;  
    }  
}
```

Overriding VS Overloading

Important note: **Overriding** a method is different from **overloading** a method!

A method (name) is **overloaded** when two or more methods have the same name, in which case the methods must differ in the number and/or types of their formal parameters (which the compiler uses to disambiguate them).

The End

- Now you can work with classes, and override the `equals` and `toString` operators