# ICS 111
# Inheritance, Object References

- Inheritance and subclasses
- Object references
- Static variables and methods

# Inheritance and Subclassing

- Review: we have seen that all exceptions are java.lang.Exception objects
  - including java.lang.RuntimeException
- this means that either of these will catch a RuntimeException:

```
catch (java.lang.Exception e)
catch (java.lang.RuntimeException e)
```

- but the second one will not catch exceptions that are not RuntimeExceptions
- that is because RuntimeException is implemented by extending Exception:

```
public class RuntimeException extends Exception {
```

- the keyword `extends` means that RuntimeException, even if it doesn't implement any public methods of its own, provides all the public methods of Exception
  - for example, RuntimeException has the printStackTrace() methods of Exception
  - in turn, Exception inherits those methods from Throwable
- we say that RuntimeException is a **subclass** of Exception:
  - all objects of type RuntimeException are also objects of type Exception, but
  - there are objects of type Exception that are not objects of type RuntimeException
- in this example, Exception is the **superclass**
- in Java, a class can only extend one superclass: Java has single inheritance

# Object Hierarchies and Implementation

- every object in Java is a subclass of Object
  - and therefore has methods `equals` and `toString` (and a few others)
  - if there is no `extends` clause in a class header, the class automatically extends Object
- an object of class X which `extends` Y stores the values of all the instance variables of both X and Y and of any of their superclasses
  - and provides all the public methods of all of these classes
- instance methods of the subclass have access to `protected` instance variables and methods of all of their superclasses:
  - methods of Y can only access the instance variables declared in Y
  - methods of X can access all the instance variables declared in X, plus any public or protected instance variables declared in Y
- and the same for methods
  - a protected method in Y is protected also in X

# Object References

- Review: we have seen that multiple variables may refer to the same underlying object
- for example:

```
ArrayList<String> a = new ArrayList<>();
ArrayList<String> b = a;
```

- Now we know more about objects, so we can understand what this really means:

```
a.add(new String("hello world");
if (b.get(b.size() - 1).equals("hello world")) { ...
```

  since b refers to the same object as a, the condition will always be true – the string "hello world" is added to both a and b by just calling a.add(), since there is only one underlying object

- similarly for arrays, and any other object that is **mutable**
  - that is, any object that has contents that can be changed

# Special Object References

- `null` is the object reference that doesn't reference any object (!!)
- `this` refers to the object that this method was called on
  - `this` is only available in instance methods and in constructors
- `super` refers to the the same object as `this`, but of the parent superclass
  - `super.method()` is how we call instance methods defined in the superclass
- `this.instanceVariable` is a clear and common way of referring to an instance variable for this object
  - `super.instanceVariable` is also clear, but much less common
- `this` is usually not required, but often improves the clarity of the code
  - e.g. the parameters to a constructor can then have the same name as the instance variables:

    ```
    this.somethingSpecial = somethingSpecial;
    ```
- `this` can also be used in calling instance methods:

  ```
  if (this.hasName()) { ...
  ```

# `this`, `super` and constructors

- one constructor can call another constructor of the same class
  - but only as the very first line of the calling constructor
  - and instead of using the class name to call the constructor, it uses `this`

```
public class Window {
  boolean windowIsOpen;
  public Window(boolean isOpen) {
    this.windowIsOpen = isOpen;
  }
  public Window() {
    this(false);
  }
}
```

- similarly, `super()` or `super(args)` is used to call a constructor of the superclass
  - it is often a good idea to have your constructor call a constructor of the superclass
  - unless you want the default constructor of the superclass to be called instead

# Object Hierarchy Example

```
public class FaceMask {
  private String faceMaskColor;
  public FaceMask(String color) {
    faceMaskColor = color;
  }
  public string getColor() {
    return faceMaskColor;
  }
}
public class AdjustableFaceMask extends FaceMask {
  private int adjustment = 0;
  public AdjustableFaceMask(String color, int adjustment) {
    super(color);                    // call the constructor of the superclass
    this.adjustment = adjustment;    // initialize our instance variable
 }
  public void adjust(int by) {
    adjustment += by;
  }
}
```

- now if I create an object of type AdjustableFaceMask, I can call its color() method:

```
AdjustableFaceMask mask = new AdjustableFaceMask("blue", 0);
String c = mask.getColor();
```

# static methods

- we have seen static methods, particularly `main`
- static methods are associated with the class, rather than with specific objects (specific instances of the class)
- example: `Math.cos()` is a pure mathematical function that computes its result based only on its parameters, and doesn't refer to any instance variables
- so it is declared as a static method
- in your code, you are welcome to make methods static when appropriate:
  - when the method doesn't use instance variables

# static variables

- there are times when we want a global variable to be shared across objects of a given class
- the book (section 8.11) has a good example: to give a distinct identifier to each object in a class
- `System.in` and `System.out` are public static variables
  - `array.length` is a public instance variable
- constants (such as `Math.PI`) are usually also declared static
- static variables can be used by code in both instance methods and static methods

# Implementation of Inheritance

- each call to `new` reserves memory for an object
- the memory must include space for:
  - all the instance variables (private, protected, or public) declared in this object
  - all the instance variables declared in the superclass, and all the way up the hierarchy
- the compiler controls access to variables and methods:
  - public means accessible to all
  - protected means accessible within the class and in all the subclasses
  - private means only accessible within the class

# Using Subclasses

- an object in a class X can be used wherever an object of its superclass Y is needed
  - e.g. in a parameter list, an assignment, or an expression
  - for example, if a method takes as parameter a type FaceMask, I can call that method with an object of type AdjustableFaceMask
- this is useful if we have true hierarchies, such as Vehicle and Car – any method that takes a Vehicle as parameter will operate on any car
  - the reverse is not true – a method that takes a Car as parameter will not operate on Vehicle objects

# Summary

- all objects are part of a hierarchy of classes rooted at Object
- the keyword `extends` is used to declare that this class is a subclass of another class
- in instance methods, `this` refers to the object the instance method was called on, and `super` to the same object but of the superclass type
- instance methods in the subclass can call protected instance methods of the superclass
- and can access protected instance variables of the superclass