The Object Class

- *Singly-rooted*. Every Java class belongs to one large inheritance hierarchy in which Object is at the top. No explicit mention of "extending" Object needs to be made in your code – it is already understood by the compiler and JVM.
- Every class has access to the following methods (and others that we will not cover here):
  - `public String toString()`
  - `public boolean equals(Object o)`
  - `public int hashCode()` [See Lesson 11]
  - `protected Object clone() throws CloneNotSupportedException`

The `toString` Method

1. If a class does not override the default implementation of `toString` given in the `Object` class, it produces output like the following:

```
public static void main(String[] args) {
    System.out.println(new Object());
    System.out.println(new StoreDirectory(null));

}

//output
java.lang.Object@18d107f
scope.more.StoreDirectory@ad3ba4
```

This is a concatenation of the fully qualified class name with the hexadecimal version of the "hash code" of the object

2. Most Java API classes override this default implementation of `toString`. The purpose of the method is to provide a (readable) `String` representation (which can be logged or printed to the console) of the state of an object.

Example from the Exercises:

```
// the Account object has this implementation of
// toString
public String toString(){
      String ret =
          "Account type: " + acctType +
                 "\nCurrent bal:  " + balance;
      return ret;
}
```

Best Practice. For every significant class you create, override the `toString` method.

3. `toString` **is automatically called when you pass an object to `System.out.println` or include it in the formation of a `String`**

4. Examples:

```
Account acct = . . . //populate an
AccountString output = "The account: " + acct;
```

```
Account acct = . . . // populate an Account
System.out.println(acct);
```

5. `toString` for arrays – sample usage

Suppose we have the array

```
String[] people = {"Bob", "Harry", "Sally"};
```

- Wrong way to form a string from an array

```
people.toString()
   //output:  [Ljava.lang.String;@19e0bfd
```

- Right way to form a string from an array

```
Arrays.toString(people)
    //output:   [Bob, Harry, Sally]
```

The `equals` Method

- Implementation in `Object` class:

`ob1.equals(ob2)` if and only if `ob1 == ob2`
if and only if references point to the same object

Using the '==' operator to compare objects is usually not what we intend (though for comparison of *primitives*, it is just what is needed). For comparing objects, the `equals` method should (usually) be overridden to compare the *states* of the objects.

Example:

```
class Person {
  private String name;
  Person(String n) {
        name = n;
  }
}
```

Here, two `Person` instances should be "equal" if they have the same `name`.
Next slide shows a good way to override `equals`

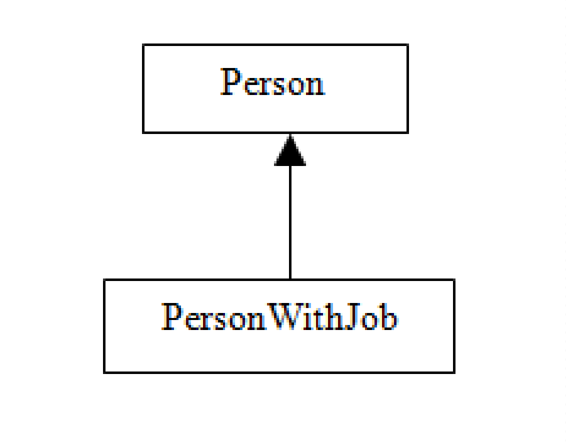
```
//an overriding equals method in the Person class
@Override
public boolean equals(Object aPerson) {
      if(aPerson == null) return false;
      if(!(aPerson instanceof Person)) return false;
      Person p = (Person)aPerson;
      boolean isEqual = this.name.equals(p.name);
      return isEqual;
} //see lesson7.equals.simple
```

Things to notice:

- The argument to `equals` must be of type `Object` (otherwise, compiler error)
- If input `aPerson` is null, it can't possibly be equal to the current instance of `Person`, so false is returned immediately
- If runtime type of `aPerson` is not `Person` (or a subclass), there is no chance of equality, so `false` is returned immediately
- After the preliminary special cases are handled, two `Person` objects are declared to be equal if and only if they have the same `name`.

Handling equals() in Inherited Classes

Example: Add a subclass `PersonWithJob` to `Person`:



```
class Person {
	private String name;
	Person(String n) {
		name = n;
	}
	public String getName() {
		return name;
	}
	@Override
	public boolean equals(Object aPerson) {
		if(aPerson == null) return false;
		if(!(aPerson instanceof Person)) return false;
		Person p = (Person)aPerson;
		boolean isEqual = this.name.equals(p.name);
		return isEqual;
	}
}
class PersonWithJob extends Person {
	private double salary;
	PersonWithJob(String n, double s) {
		super(n);
		salary = s;
	}
}
```

The `equals()` method is inherited by `PersonWithJob` in this implementation. So objects of type `PersonWithJob` are compared only on the basis of the name field.

Example:

```
PersonWithJob joe1 = new PersonWithJob("Joe", 100000);
PersonWithJob joe2 = new PersonWithJob("Joe", 50000);
boolean areTheyEqual = joe1.equals(joe2);  //areTheyEqual
                                                == true
```

Best Practices: If, in your code, this kind of situation does not present a problem – if it is OK to inherit `equals()` in this way – then the implementation given here is optimal. This is called the *instanceof strategy for overriding equals*

Best practice in the case where subclasses need to have their own form of equals is more complicated (discussed below)

What Happens When Subclasses Need Their Own Form of equals()

Example. Provide `PersonWithJob` **its own** `equals` **method.**

Demo: `lesson7.equals.asymmetry`

```
//an overriding equals method in the PersonWithJob class
@Override
public boolean equals(Object withJob) {
       if(withJob == null) return false;
       if(!(withJob instanceof PersonWithJob))
              return false;
       PersonWithJob p = (PersonWithJob)withJob;
       boolean isEqual= getName().equals(p.getName()) &&
                          this.salary == p.salary;
       return isEqual;
}
```

This creates a serious problem, called *asymmetry* (violates contract for equality)

```
Person p = new Person("Joe");
PersonWithJob withJob = new PersonWithJob("Joe",
                                          100000);
//true - using Person’s equals()
theyreEqual1 = p.equals(withJob);

//false - using PersonWithJob’s equals()
theyreEqual2 = withJob.equals(p);
```

Example. Implement `equals` in a different way for both `Person` and `PersonWithJob`.

Demo: `lesson7.equals.sameclass`

```
//alternative equals method in the Person class
@Override
public boolean equals(Object aPerson) {
    if(aPerson == null) return false;
    if(aPerson.getClass() != this.getClass())
           return false;
    Person p = (Person)aPerson;
    boolean isEqual = this.name.equals(p.name);
    return isEqual;
}

 //alternative equals method in the PersonWithJob class
@Override
public boolean equals(Object withJob) {
    if(withJob == null) return false;
    if(withJob.getClass() != this.getClass())
           return false;
    PersonWithJob p = (PersonWithJob)withJob;
    boolean isEqual = getName().equals(p.getName()) &&
                        this.salary == p.salary;
    return isEqual;
}
```

This solves the asymmetry problem – now, it is impossible for a `PersonWithJob` object to be equal to a `Person` object, using either of the `equals()` methods. This is called the *same classes strategy for overriding equals*.

Difficulty with the Same Classes Strategy

Example: Continuing the example from above, suppose we have a subclass `PersonWithJobWithCounter` of `PersonWithJob`.

Demo: `lesson7.equals.sameclass.theproblem`

```
  //using same classes strategy as shown earlier
class PersonWithJob extends Person {
    private double salary;
    PersonWithJob(String n, double s) {
        super(n);
        salary = s;
    }
    @Override
    public boolean equals(Object withJob) {
        if(withJob == null) return false;
        if(withJob.getClass() != this.getClass())
            return false;
        PersonWithJob p = (PersonWithJob)withJob;
        boolean isEqual =
            getName().equals(p.getName()) &&
            this.salary == p.salary;
        return isEqual;
    }
}
```

```
class PersonWithJobWithCounter
                        extends PersonWithJob {
     static private int counter;
     PersonWithJobWithCounter(String n, double s){
           super(n, s);
           counter++;
     }
}
```

The intention here is that `PersonWithJobWithCounter` **will use the** `equals` **method of its superclass. But this creates a problem:**

```
     PersonWithJob joe1 =
           new PersonWithJob("Joe", 50000);
     PersonWithJobWithCounter joe2 = new
           PersonWithJobWithCounter("Joe", 50000);
     joe2.equals(joe1); //value is false since joe2
                         //is not of same type as joe1
```

Best Practice When Using Same Classes Strategy

The example shows that whenever the same classes strategy is used to provide separate `equals` methods for classes B and A, where B is a subclass of A, then we should prevent the possibility of creating a subclass of B to prevent the introduction of unexpected results of `equals` method.

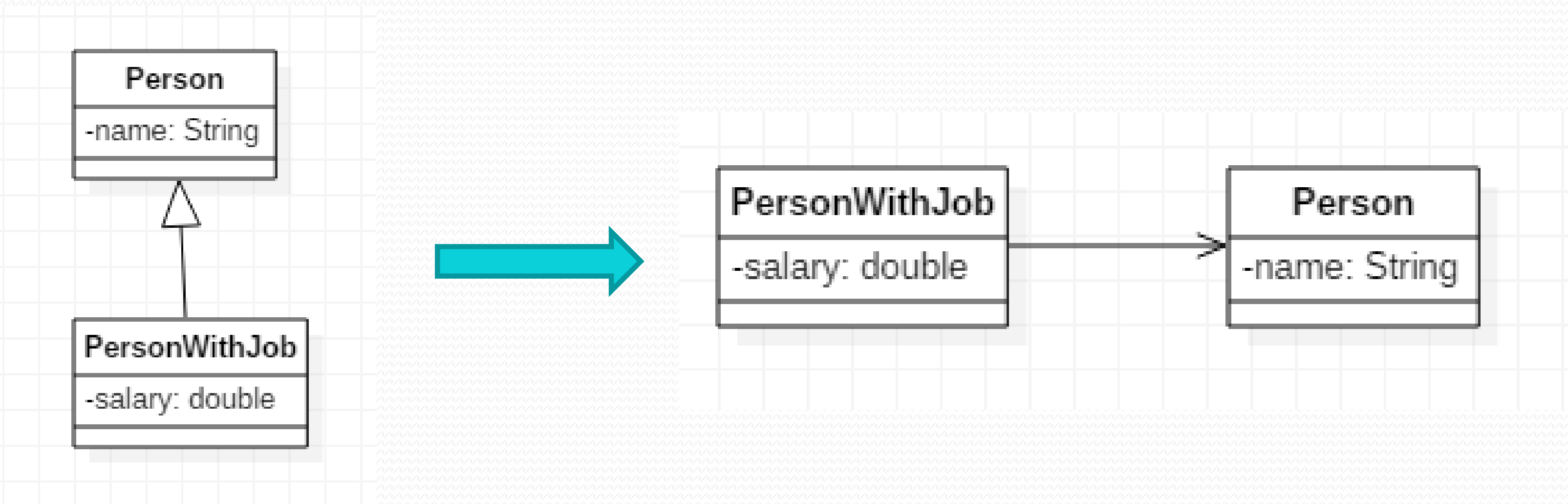
Best Practice – Same Classes Strategy. If B is a subclass of A and each class has its own `equals` method, implemented using the same classes strategy, then the class B should be declared `final` to prevent unexpected result of applying `equals` in a subclass of B.

Question. What if we don't wish to make B `final`?

Using Composition Instead of Inheritance

Previous example shows that when a class B inherits from a class A and you want B and A to have different equals methods, you need to use the same classes strategy and you have to make B final. This procedure is somewhat error-prone (you may forget to make B final) and may not even be desirable (your design may require you to allow B to have subclasses).

An alternative to the same classes strategy, when you need classes A and B to have separate equals methods, is *to use composition instead of inheritance*. In that case, B does *not* inherit from A. This is always a good alternative unless you are needing to use polymorphism.



Example: *Implementing Manager using Composition instead of Inheritance* (See sample code in package `lesson7.lecture.empmanager.usecomposition`)

Summary of Best Practices for Overriding Equals In the Context of Inheritance

Suppose B is a subclass of A.

- If it is acceptable for B to use the same equals method as used in A, then the best strategy is the *instanceof strategy*

- If *two different equals methods* are required, two strategies are possible
  - Use the same classes strategy, but declare subclass B to be final
  - Use composition instead of inheritance – this will always work as long as the inheritance relationship between B and A is not needed (e.g. for polymorphism)

Overriding the hashCode() Method

- Any implementation of the Hashtable ADT in Java will make use of the `hashCode()` function as the first step in producing a hash value (or table index) for an object that is being used as a key.
- Default implementation of `hashCode()` provided in the `Object` class is not generally useful – gives a numeric representation of the memory location of an object. See `lesson7.lecture.hashcode.bad1`
- **Example**: We wish to use pairs (firstName, lastName) as keys for Person objects in a hashtable.
(See package `lesson7.lecture.hashcode.bad2`)

Demo shows default `hashCode` method is not useful. If two `Pair` objects, created at different times, are equal (using the `equals` method), we would expect them to have the same `hashCodes`, so that, after hashing, they are sent to the same table slot. But default `hashCode` method does not take into account the fields used by `equals` method, so equal `Pair` objects may be assigned different slots in the table.

hashCode() Rules

- To use an object as a key in hashtable, you must override `equals()` and `hashCode()`
- (Primary Hashing Rule) If k_1, k_2 are keys and k_1`.equals(`k_2`)` then it must be true that [k_1`.hashCode() ==` k_2`.hashCode()`]
  This means that you must include the same information in your `hashCode` definition as you include in your implementation of `equals`.

Creating Good Hash Codes When Overriding hashCode()

- There are two general rules for creating hash codes:
  I. (Primary Hashing Rule) Equal keys must be given the same hash code (otherwise, the same key will occupy different slots in the table)

     If `k1.equals(k2) then k1.hashCode() == k2.hashCode()`

  II. (Secondary Hashing Guideline) Different keys should be given different hash codes (if not, in the worst case, if every key is given the same hash code, then all keys are sent to the same slot in the table; in this case, hashtable performance degrades dramatically).

     Best Practice: The hash codes should be distributed as evenly as possible (this means that one integer occurs as a hash code approximately just as frequently as any other)

Creating Hash Codes from Object Data (Legacy Approach)

You are trying to define hashCode() for your class and you want to build the hash code from the hash codes of each variable in the class. First step is to assign hash code values to each of these instance variables.

```
class MyClass {
      boolean val;
      double x;
      String name;
        . . .
      public int hashCode() {
            //implement
      }
}
```

Suppose `f` is an instance variable in your class

- If `f` is boolean, compute `(f ? 1 : 0)`
- If `f` is a byte, char, short, or int, compute `(int) f`.
- If `f` is a long, compute `(int) (f ^ (f >>> 32))`
- If `f` is a float, compute `Float.floatToIntBits(f)`
- If `f` is a double, compute `Double.doubleToLongBits(f)` which produces a long `f1`, then return `(int) (f1 ^ (f1 >>> 32))`
- If `f` is an object, use `f.hashCode()` (best if implementation of `f` has overridden `hashCode()` already)

Combining HashCodes of Instance Variables to Produce a Final HashCode (Legacy Approach)

- Suppose your class has instance variables u, v, w and corresponding hashCodes hash_u, hash_v, hash_w (obtained as in the previous slide). Then:

```
@Override
public int hashCode() {
  int result = 17;
  result += 31 * result + hash_u;
  result += 31 * result + hash_v;
  result += 31 * result + hash_w;
  return result;
}
```

Combining HashCodes of Instance Variables to Produce a Final HashCode (Modern Approach)

- Use the following method in the `Objects` class to compute hash code.

```
public static int hash(Object... values)
```

- For example, if an object has three fields, x, y, and z, we could compute hash code in this way:

```
@Override
public int hashCode() {
  return Objects.hash(x, y, z);
}
```

Overriding HashCode for Pair class

Example. *Overriding hashCode in the Person-Pair example.* We must take in account the same fields in computing `hashCode` as those used in overriding `equals`. The fields in `Pair` are `String`s, and Java already provides `hashCodes` for `Strings`. So we make use of these and combine them to produce a complex `hashCode` for `Pair`. (See the solution in `lesson7.lecture.hashcode.bad2.fix`)

```java
//modern way
public int hashCode() {
    return Objects.hash(first, second);

}
```

```java
//legacy approach
public int hashCode() {
    int result = 17; //seed
    int hashFirst = first.hashCode();
    int hashSecond = second.hashCode();
    result += 31 * result + hashFirst;
    result += 31 * result + hashSecond;
    return result;

}
```