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()`

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 Person("John Smith"));
}

//output
java.lang.Object@18d107f
com.example.Person@ad3ba4
```

This is a concatenation of the fully qualified class name with the hexadecimal version of the "hash code" of the object (we will discuss hash codes later in this set of slides)

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 newline =
          System.getProperty("line.separator");
      String ret =
          "Account type: " + acctType + newline +
                  "Current 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;
  }
}
```

Two `Person` instances should be "equal" if they have the same name? OR SSN? AND birth date?

```
//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;
}
```

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.

Overriding hashCode()

- HashMap depends on hashCode() method and two objects are considered equal if they have equal hashCode()
- hashCode() default implementation calculate hash code based on object reference.
- So, for HashMap to work properly, we need to follow following rule

HashCode Rules

1. *Whenever equals is overridden in a class, hashCode must also be overridden*
2. *The hashCode method must take into account the same fields as those that are referenced in the overriding equals method*.

Comparing Objects for Sorting and Searching

- Java supports sorting of many types of objects. To sort a list of objects, it is necessary to have some “ordering” on the objects. For example, there is a natural ordering on numbers and on `Strings`. But what about a list of `Employee` objects?
- In practice, we may want to sort business objects in different ways. An `Employee` list could be sorted by name, salary or hire date.

Employee Data		
Name	Hire Date	Salary
Joe Smith	11/23/2000	50000
Susan Randolph	2/14/2002	60000
Ronald Richards	1/1/2005	70000

Make Comparable

- One way to accomplish this is making the objects comparable by implementing interface Comparable.
- **Comparable** interface, whose only method is **`compareTo()`**. Like lists, in j2se5.0, `Comparable` are <parametrized>
- The `compareTo()` method is expected to behave in the following way (so it can be used in conjunction with the Collections API):

  For `this` object being compared and the other object `other`,
  - `compareTo(other)` returns a negative number if this is “less than” `other`
  - `compareTo(other)` returns a positive number if this is “greater than” `other`
  - `compareTo(other)` returns 0 if this “equals” `other`

Use Comparator

- To accomplish this, you specify your own ordering on a class using the **Comparator** interface, whose only method is **compare()**. Like lists, in j2se5.0, `Comparators` are <parametrized>.

- The `compare()` method is expected to behave in the following way (so it can be used in conjunction with the Collections API):

  For objects a and b,
  - `compare(a,b)` returns a negative number if a is “less than” b
  - `compare(a,b)` returns a positive number if a is “greater than” b
  - `compare(a,b)` returns 0 if a “equals” b