The Object Class

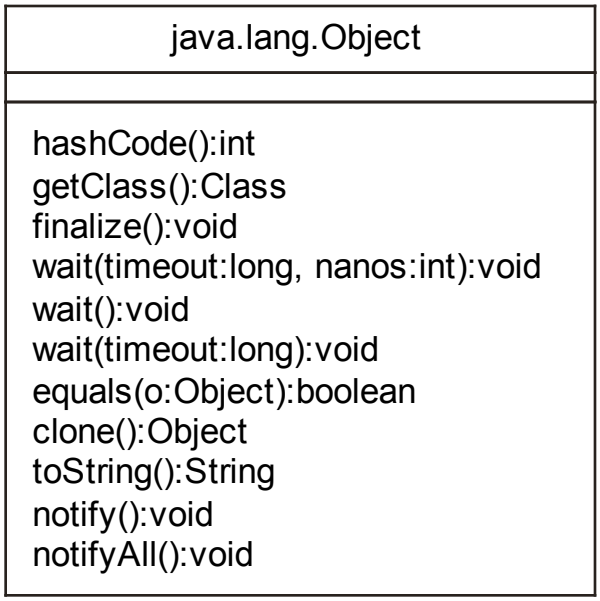
- Overriding toString()
- Overriding equals() and hashCode()
- Overriding clone()

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Object Class

- Object is the ultimate superclass of all Java classes
- Object thus defines the minimal behavior for all Java objects



The Object class is the root of all class hierarchies in Java and provides a basic set of behavior that's common across all Java objects.

In this chapter, we will examine a few of those behaviors in more detail.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The toString() Method

- Many Java methods (e.g. System.out.println) call an object's toString() method to obtain a String representation of the object
- You can override toString() to display the object's state for debugging and other purposes

```
1    Student s1 = new Student (12, "Bill", 3.44);
2
3    System.out.println(s1);
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Overriding toString()

```
1    public class Student implements Cloneable
2    {
3        private int studentID;
4        private String name;
5        private double gpa;
6        private Advisor advisor =
7            new Advisor("<no-name>", 0);
8        . . .
9        public String toString()
10       {
11           StringBuffer sb = new StringBuffer();
12           sb.append ("ID: " + studentID);
13           sb.append (" Name: " + name);
14           sb.append (" GPA: " + gpa);
15           sb.append (" Advisor name: " +
16               advisor.getName());
17           return sb.toString();
18       }
19   }
```

A common use of toString() is to display the current state of the object for debugging and diagnostic purposes. Here we override toString() and use a StringBuffer to return a formatted String with the Student's data.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The equals() Method

- Recall that the Java equality operator (==) when applied to references, compares references, not the contents of the object to which they refer
- The implementation of equals() in Object compares references
- You can override equals() and thus determine what it means for objects of your class to be equivalent

```
1    Student s1 = new Student (12, "Bill", 3.44);
2    Student s2 = new Student (12, "Bill", 3.44);
3
4    boolean b1 = (s1 == s2);     // false
5    boolean b2 = s1.equals(s2);  // true
```

You can find a model for this sort of behavior in the String class, which overrides equals() and returns true only if the characters in the two Strings are identical.

Note that you call equals() on one object, passing a reference to the other.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Overriding equals()

```
1    public class Student implements Cloneable
2    {
3        private int studentID;
4        private String name;
5        private double gpa;
6        private Advisor advisor =
7            new Advisor("<no-name>", 0);
8
9        public boolean equals(Object obj)
10       {
11           if (obj == this)
12               return true;
13
14           if (!(obj instanceof Student))
15               return false;
```

Starting a listing of an overriden equals() method, we first do a quick test to see if the other object is in fact the same object as this object. While this test is not strictly necessary, it's a performance enhancement that allows to skip the other testing.

We then check to see if the other object is of type Student -- if not, then it cannot be equal.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Overriding equals(), cont'd

```
16            Student other = (Student)obj;
17            if ( this.studentID == other.studentID &&
18                 this.name.equals(other.name))
19                return true;
20            else
21                return false;
22        }
23    . . .
```


Continuing the equals() method, once we know that the other object is a Student, we can safely cast it's reference and then determine if the other object is equivalent by comparing its fields with the current object's fields.

Whenever you override equals(), you need to decide what it means for objects of the class to be the same. In this case, we decided that if two Students have the same ID AND the same name, then they are equivalent. Note that this decision depends on your object model -- in a different Student model, it might be enough for the IDs to be the same or for all of the Student fields to be the same.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The hashCode() Method

- Some collections (e.g. HashMap) depend on an object being able to generate a pseudo-unique integer based on the object's state
- You can override hashCode() to provide that integer
- For consistency, objects that are equal() must have the same hashcode

```
1    HashMap<Student, String> hm = new HashMap<>();
2    Student s1 = new Student (12, "Bill", 3.44);
3    Student s2 = new Student (44, "Sue", 2.35);
4    Student s3 = new Student (55, "Harry", 3.59);
5    hm.put(s1, "Bill's String");
6    hm.put(s2, "Sue's String");
7    hm.put(s3, "Harry's String");
```

Hash table-based collections depend on using an integer, derived from the key, for fast lookup. For best results, the integer should be different for most (if not all) unequal objects and should be distributed evenly. Objects that are considered the same by the equals() method must have the same hashcode, but the reverse is not necessarily true (i.e. two unequal objects can have the same hashcode). From:

http://www.math.utah.edu/~beebe/software/hash/hash.html

Hash tables, sometimes also known as associative arrays, or scatter tables, are a data structure that offers fast lookup, insertion, and deletion of (key, value) pairs. In a well-designed implementation of hash tables, all of these operations have a time complexity of O(1), rather than the O(n) that linked lists, association lists, and vectors would require for some of them.

Hashing was discovered and first published and implemented on computers in the early 1950s, and is based on a simple idea: transform the key to a number (the hash process), and then use that number to index a table. In order to keep the table size manageable, reduce the computed number modulo the table size.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Rules for Implementing hashCode()

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

This contract is from the JavaDoc for the Object class.

The key point is the second, which implies that if you override equals(), you should also override hashCode() and ensure that the hash code is calculated using the same fields that equals() checks for equivalence.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Overriding hashCode()

```
1    public class Student implements Cloneable
2    {
3        private int studentID;
4        private String name;
5        private double gpa;
6        private Advisor advisor =
7            new Advisor("<no-name>", 0);
8
9        public int hashCode()
10       {
11           int result = 17;
12
13           result = 37*result + studentID;
14           result = 37*result + name.hashCode();
15           return result;
16       }
17   . . .
18 }
```

Here we show a simple, but functional implementation of hashCode(). Note that the code is calculated based on the Student's ID and name, which are the same two fields that we checked in the equals() method.

This recipe that uses prime numbers to generate a well distributed hashcode is taken from the book "Effective Java Programming Language Guide" by Joshua Bloch. The chapter from which this recipe is taken is available for download as a PDF from:

http://java.sun.com/docs/books/effective/chapters.html

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Cloning Objects

- Sometimes it's useful to make a copy of an object rather copying the object's reference
- Java provides the clone() method for this purpose

```
1    Student s1 = new Student (12, "Bill", 3.44);
2    s1.setAdvisorName("Cindy");
3
4    try
5    {
6        Student s2 = (Student)s1.clone();
7        s2.setAdvisorName("Sue");
8        System.out.println(s1);
9        System.out.println(s2);
10   }
11   catch (CloneNotSupportedException e)
12   {
13        e.printStackTrace();
14   }
```

The idea behind clone() is that you want to create a new object that has the same state as an existing one.

Note that there are other ways to accomplish copying without using clone(). For example, you could write a so-called "copy constructor", which is a constructor which accepts a parameter of an object of the same type.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Implementing the Cloneable Interface

- The Cloneable interface is a tagging interface that changes the behavior of the clone() method in the Object class
- If a class does not implement Cloneable, then the clone() method in Object throws the CloneNotSupportedException

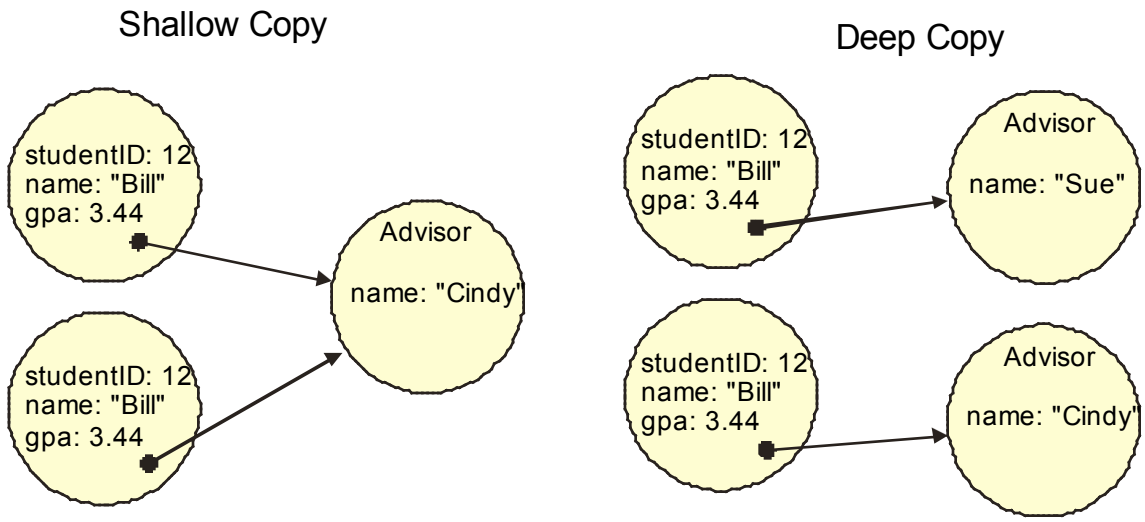
```
1    public class Student implements Cloneable
2    {
3       . . .
4    }
```

The Cloneable interface has no methods. Implementing it changes the behavior of the clone() method define in Object. We will see how this works in a moment, but note that the code on the last page catches this exception.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Danger of Shallow Copying

- If you are not careful, you can end up with a cloned object that is still tied to the original (i.e. changes to either the clone or original affects the other)



The Student class maintains a reference to its Advisor, which is a mutable object. The Student class also provides a method to change the Advisor. Please refer back to the code shown two pages ago.

In that code, we create a Student named "Bill" and set the Student's advisor's name to "Cindy". We then clone the Student and change the clone's advisor name to "Sue".

The cloned object should be a new, independent object, so changing the clone's advisor name should NOT affect the original Student. However, that's what will happen if the Student close implements a shallow copy.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Overriding clone()

```
1    public class Student implements Cloneable
2    {
3        private int studentID;
4        private String name;
5        private double gpa;
6        private Advisor advisor =
7            new Advisor("<no-name>", 0);
8
9        public Object clone()
10           throws CloneNotSupportedException
11       {
12           Student s = (Student)super.clone();
13           s.advisor = (Advisor)this.advisor.clone();
14
15           return s;
16       }
17   . . .
18   }
```

This listing shows how the Student class can override clone(), performing a deep copy as required.

In line 12, we call the Object implementation of clone(), which creates a new object and performs a bit-by-bit shallow copy. The shallow copy is perfectly OK for all fields that are primitives or references to immutable objects, which includes the name, studentID and gpa fields (remember that Strings are immutable!).

The only reference-typed field that refers to a mutable object is Advisor, so in line 13, we overwrite the shallow reference copy by telling the Advisor object to clone itself, which creates a new, cloned Advisor object.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Overriding clone(), cont'd

```
1    public class Advisor implements Cloneable
2    {
3        private String name;
4        private double salary;
5
6        public Object clone()
7            throws CloneNotSupportedException
8        {
9            Advisor a = (Advisor)super.clone();
10
11           return a;
12       }
13   . . .
14   }
```


Here we show part of the Advisor class. Note that since it only has primitive and immutable-object references as fields, the clone() method can simply use Object's shallow, bit-by-bit copy.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Review Questions

- Why is it important that if you override equals(), you also override hashCode()?
- Explain why a shallow copy is dangerous.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About the toString(), equals(), hashCode() and clone() methods from the Object class

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum