04 The *Object* Class

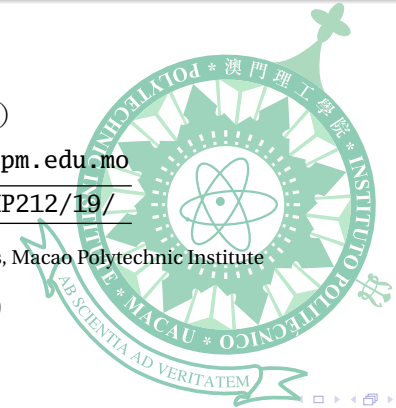
Instructor: Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

September 19 (24), 2019



Outline

Methods Common to All Objects

- Although *Object* is a concrete class, it is designed primarily for extension.
- All of its methods have explicit *general contracts* because they are designed to be overridden.

```
public    boolean  equals(Object obj)
public    int      hashCode()
public    String   toString()
```

- It is the responsibility of any class overriding these methods to obey their general contracts; failure to do so will prevent other classes that depend on the contracts (such as *HashMap* and *HashSet*) from functioning properly in conjunction with the class.
- While not an *Object* method, the interface method of *Comparable*$\langle T \rangle$, `int compareTo(T y)`, also has a similar character.

When Not to Override *equals*

- The *equals* method for class *Object* implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object.
- Overriding the equals method seems simple, but there are many ways to get it wrong. The easiest way to avoid problems is *not* to override the *equals* method.
- When each instance of the class is inherently unique, such as *Thread* that represent active entities rather than values.
- When we don't care whether the class provides a "content equality" test.
- When a superclass has already overridden *equals*, and the superclass behavior is appropriate for this class.
- When the class is private or package-private, and you are certain that its equals method will never be invoked.

How to Override *equals*

- We override *equals* when a class has a notion of content equality that differs from mere object identity, and a superclass has not already overridden equals to implement the desired behavior. This is common for value classes, such as *Rational* numbers.
- The *equals* method implements an *equivalence* relation. It is:
  - **Reflexive**: For any non-null reference value x, $x.equals(x)$ must return `true`.
  - **Symmetric**: For any non-null reference values x and y, $x.equals(y)$ must return the same as what $y.equals(x)$ returns.
  - **Transitive**: For any non-null reference values x, y, z, if $x.equals(y)$ returns `true` and $y.equals(z)$ returns `true`, then $x.equals(z)$ must return `true`.

  Plus:
  - For any non-null reference values x and y, multiple invocations of $x.equals(y)$ *consistently* return `true` or *consistently* return `false`, provided x and y are not changed.
  - For any non-null reference value x, $x.equals(\texttt{null})$ must return `false`.

Example: Overriding *equals* in the *Rational* Class

```
1  public class Rational {
2      ...
3      @Override
4      public boolean equals(Object obj) {
5          if ( this == obj ) return true;   // an optimization
6          else if ( obj instanceof Rational ) {
7              Rational y = (Rational)obj;
8              return num.equals(y.num) && denom.equals(y.denom);
9          } else return false;
10     }
11 }
```

Example: A Violation of Symmetry

```
1  public class Rational {
2      ...
3      @Override public boolean equals(Object obj) {
4          if ( this == obj ) return true;
5          else if ( obj instanceof Rational ) {
6              Rational y = (Rational)obj;
7              return num.equals(y.num) && denom.equals(y.denom);
8          } else if ( obj instanceof BigInteger ) {
9          // One-way interoperability!
10             BigInteger y = (BigInteger)obj;
11             return num.equals(y) && denom.equals(BigInteger.ONE);
12         } else return false;
13     }
14 }
```

The Problem of Subclass *equals*

- The problem arises when we introduce a new field in the subclass, and try to extend the *equals* method to include this field.

```
class ColorPoint extends Point { int color; ... }
```

- If we include the *color* field, and consider a *ColorPoint* not equal to a *Point*, then it will violate the symmetry, since a *Point* regards a *ColorPoint* as a *Point* and can possibly return `true` from its *equals*.
- If we include the *color* field only when comparing two *ColorPoint*s, and ignore the *color* field when comparing a *ColorPoint* to a *Point*, then we preserve the symmetry.
- However, by the above approach, we violate the transitivity: when a *ColorPoint a* equals a *Point b*, and *b* equals another *ColorPoint c*, with both comparisons ignoring the *color* field, we cannot make sure that *a* equals *c*, with the consideration of the *color* field.
- There is no way to extend a class that has instances, and add a field to consider in *equals* while preserving the contract.

Always Override *hashCode* When You Override *equals*

- Equal objects must have equal hash codes.
- Failure to maintain the above property will prevent a class from functioning properly in conjunction with all hash-based collections, including *HashMap*, *HashSet* and *Hashtable*.
- Two distinct instances may be content equal according to a class's *equals* method, but to *Object*'s *hashCode* method, they are just two objects with nothing much in common. *Object*'s *hashCode* method returns two seemingly random numbers instead of two equal numbers.
- Therefore, we need to override the *hashCode* method to match the overridden *equals*.

A Simple Recipe to Compute Hash Codes for Objects

a. Store some constant nonzero value, say, 17, in an `int` variable called *result*.
b. Compute an `int` hash code *c* for each of the significant fields *f*:
   1. For a `boolean`, compute (*f* `? 1 : 0`).
   2. For a `byte`, `char`, `short`, or `int`, compute `(int)`*f*.
   3. For a `long`, compute `(int)(`*f* `^ (`*f* `>>> 32))`.
   4. For a `float`, compute *Float*.*floatToIntBits*`(` *f* `)`.
   5. For a `double`, compute *Double*.*doubleToLongBits*`(` *f* `)`, and then hash the resulting `long` as in step 3.
   6. For an object reference, recursively invoke *f*.*hashCode*`()` on the field. If the value of the field is `null`, return 0.
c. Combine each of the hash codes *c* into *result* as follows:

$$result = 31 * result + c;$$

Example: *hashCode* of the *Rational* Class

```
1  public class Rational {
2      private BigInteger num, denom;
3      ...
4      @Override
5      public int hashCode() {
6          int result = 17;
7          result = result*31+num.hashCode();
8          result = result*31+denom.hashCode();
9          return result;
10     }
11 }
```

If a class is immutable and the cost of computing the hash code is significant, we may consider caching the hash code in the object rather than recalculating it each time it is requested.

Example: Caching Hash Codes

```
1  public class Rational {
2      private BigInteger num, denom;
3      private volatile int hc = 0;  ...
4      @Override
5      public int hashCode() {  // Lazily initialized, cached hashCode
6          int result = hc;
7          if ( result == 0 ) {
8              result = 17;
9              result = result*31+num.hashCode();
10             result = result*31+denom.hashCode();
11             hc = result;
12         }
13         return result;
14     }
15 }
```

Always Override *toString*

- Providing a good *toString* implementation makes your class much more pleasant to use.
- The *toString* method is automatically invoked when an object is passed to *println*, *printf*, the string concatenation operator, or *assert*, or printed by a debugger.
- When practical, the *toString* method should return all of the interesting information contained in the object.
- It's usually a good idea to provide a matching static factory or constructor so programmers can easily translate back and forth between the object and its string representation.
- Provide programmatic access to all of the information contained in the value returned by *toString*. If you fail to do this, you *force* programmers who need this information to parse the string.

Example: *toString* of the *Rational* Class

```
1 public class Rational {
2     private BigInteger num, denom;
3     ...
4     @Override
5     public String toString() {
6         return num.toString()+" / "+denom.toString();
7     }
8 }
```

Example: Constructors vs. Static Factory Methods

The following is a problematic constructor to translate strings to *Rational* Numbers.

```
1  public class Rational {
2      private BigInteger num, denom; ...
3      public Rational(String s) {
4          int p = s.indexOf('/');
5          if ( p == -1 ) {
6              num = new BigInteger(s.trim());
7              denom = BigInteger.ONE;
8          } else {
9              num = new BigInteger(s.substring(0, p).trim());
10             denom = new BigInteger(s.substring(p+1).trim());
11         }
12     }
13 }
```

Example: Constructors vs. Static Factory Methods

The following is a problematic constructor to translate strings to *Rational* Numbers.

```
1  public class Rational {
2      private BigInteger num, denom; ...
3      public Rational(String s) {
4          int p = s.indexOf('/');
5          if ( p == -1 ) {
6              num = new BigInteger(s.trim());
7              denom = BigInteger.ONE;
8          } else {
9              num = new BigInteger(s.substring(0, p).trim());
10             denom = new BigInteger(s.substring(p+1).trim());
11         }
12     }
13 }
```

Write a static factory instead, or put the normalization in an ordinary private method.

Consider Implementing *Comparable*

- Unlike the other methods discussed, the *compareTo* method is not declared in Object. Rather, it is the sole method in the *Comparable* interface.
- By implementing *Comparable*, you allow your class to interoperate with all of the many generic algorithms and collection implementations that depend on this interface.
- If you are writing a value class with an obvious natural ordering, such as alphabetical order, numerical order, or chronological order, you should strongly consider implementing the interface.

```
public interface Comparable<T> {
    int compareTo(T y);
}
```

- The *compareTo* method compares this object with the specified object for order and returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Contract of the *compareTo* Method

- $x.compareTo(y)$ `>` 0 if and only if $y.compareTo(x)$ `<` 0.
- $x.compareTo(y)$ `==` 0 if and only if $y.compareTo(x)$ `==` 0.
- (transitivity) $x.compareTo(y)$ `>` 0 `&&` $y.compareTo(z)$ `>` 0 implies $x.compareTo(z)$ `>` 0.
- $x.compareTo(y)$ `==` 0 implies that $x.compareTo(z)$ agrees with $y.compareTo(z)$.
- It is strongly recommended, but not strictly required, that `(`$x.compareTo(y)$ `==` 0`)` `==` `(`$x.equals(y)$`)`.

Example: *compareTo* of the *Rational* Class

```
1  public class Rational implements Comparable<Rational> {
2      private BigInteger num, denom;
3      ...
4      @Override
5      public int compareTo(Rational y) {
6          return  num.multiply(y.denom).compareTo(
7                  denom.multiply(y.num));
8      }
9  }
```