Effective Java, Chapter 3

Methods Common to All Objects

Agenda

- Material From Joshua Bloch
  - Effective Java: Programming Language Guide
- Cover Items 10 through 14
  - Methods Common to All Objects

Item 10 equals

- Obey the general contract when overriding `equals()`
- Overriding seems simple, but there are many ways to get it wrong.
- Best approach – Avoid to override the `equals` method!

When NOT to override *equals*

- Best approach – Avoid! Works if:
  - Each instance of a class is unique
    - Enum
    - Represents active entities rather than values, e.g. Thread
  - You don't care if class has logical equality
    - E.g. no need to override *equals* method of Random class
  - The superclass equals is satisfactory
    - equals implementation of AbstractSet/AbstractList/AbstractMap
  - Class is not public and equals never used

General contract for equals

- Reflexive
  - `x.equals(x)` must be true
- Symmetric
  - `x.equals(y)` iff `y.equals(x)`
- Transitive
  - If `x.equals(y) && y.equals(z)`
  - Then `x.equals(z)`
- Consistency…
  - Multiple invocation return the same result, provided no information used in equals comparisons on the objects is modified.
- Null values:
  - `x.equals(null)` is always false

How hard could this be?

- Reflexivity is pretty much automatic
- Symmetry is not:
  - Example `CaseInsensitiveString`

```
private String s;

@Override public boolean equals (Object o) {
   if (o instanceof CaseInsensitiveString)
      return s.equalsIgnoreCase(
         ((CaseInsensitiveString) o).s);
   if (o instance of String)
      return s.equalsIgnoreCase((String) o);
   return false;
}
```

How hard could this be?

- Reflexivity is pretty much automatic
- Symmetry is not:
  - Example `CaseInsensitiveString`

```
private String s;
// Broken - violates symmetry
@Override public boolean equals (Object o) {
   if (o instanceof CaseInsensitiveString)
      return s.equalsIgnoreCase(
         ((CaseInsensitiveString) o).s);
   if (o instance of String) // Not Symmetric!
      return s.equalsIgnoreCase((String) o);
   return false;
}
```

Why does this violate symmetry?

- Consider this code:

```
Object x = new CaseInsenstiveString (“abc”);
Object y = “Abc”;  // y is a String
if (x.equals(y))  {…}  // evaluates true, so execute
if (y.equals(x))  {…}  // evaluates false, so don’t…
```

- Dispatching of `equals()` calls
  - First `equals()` call to `CaseInsensitiveString`
  - Second `equals()` call to `String`
- This is horrible!

Correct Implementation

- Avoid temptation to be “compatible” with the `String` class:

```
// CaseInsensitiveString is not a subclass of String!
private String s;
@Override public boolean equals (Object o) {
   return (o instanceof CaseInsensitiveString)
      &&
      (CaseInsensitiveString o).s.
      equalsIgnoreCase(s);
}
```

Symmetry and Transitivity

- Surprisingly difficult – general result about inheritance
- Example:
  - A 2D `Point` class
    - State is two integer values x and y
    - `equals()` simply compares x and y values
  - An extension to include color
    - public class `ColorPoint` extends `Point`
    - What should `equals()` do?

Preliminaries: What does equals in Point look like?

```
public class Point {  // routine code
   private int x; private int y;
   ...
   @Override public boolean equals(Object o) {
      if (!(o instanceof Point))
         return false;
      Point p = (Point) o;
      return p.x == x && p.y == y;
   }
}
```

Choice 1 for equals() in ColorPoint

- Have equals() return true iff the other point is also a ColorPoint:

```
// Broken - violates symmetry
 @Override public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
       return false;
    ColorPoint cp = (ColorPoint) o;
    return super.equals(o) &&
       cp.color == color;
 }
```

Problem

- Symmetry is broken
- Different results if comparing:

```
ColorPoint cp = new ColorPoint (1, 2, RED);
Point p = new Point (1,2);
// p.equals(cp), cp.equals(p) differ
```

- Unfortunately, `equals()` in `Point` doesn't know about `ColorPoints`
  - Nor should it…
- So, try a different approach…

Choice 2 for equals() in ColorPoint

- Have `equals()` ignore color when doing “mixed” comparisons:

```
// Broken - violates transitivity (ColorPoint)
@Override public boolean equals(Object o) {
    if (!(o instance of Point)) return false;
    // If o is a normal Point, be colorblind
    if (!o instanceof ColorPoint)
       return o.equals(this);
    ColorPoint cp = (ColorPoint) o;
    return super.equals(o) && cp.color == color;
 }
```

Now symmetric, but not transitive!

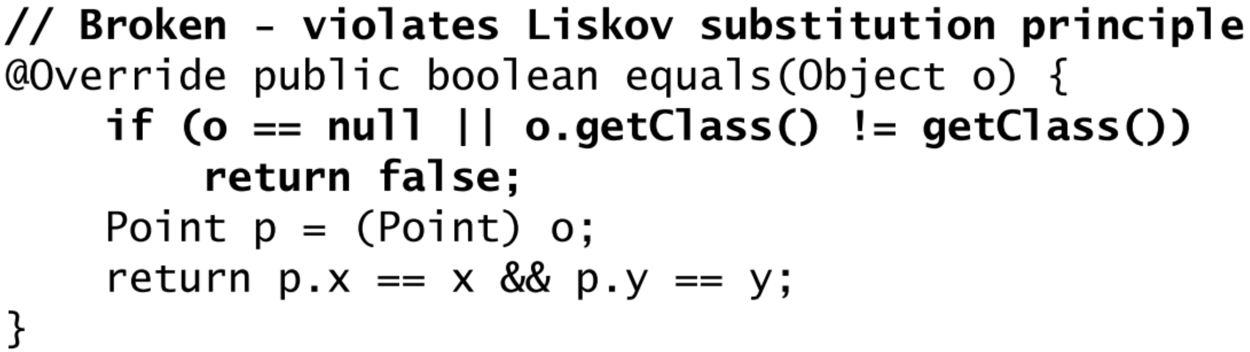
- Consider the following example

```
ColorPoint p1 = new ColorPoint(1,2,RED);
Point p2 = new Point(1,2);
ColorPoint p3 = new ColorPoint(1,2,BLUE);
```

- The following are true:
  - `p1.equals(p2)`
  - `p2.equals(p3)`
- But not `p1.equals(p3)`!

How about *getClass*?

- Use *getClass* insteadof *instanceof*?



```
// Broken - violates Liskov substitution principle
@Override public boolean equals(Object o) {
    if (o == null || o.getClass() != getClass())
        return false;
    Point p = (Point) o;
    return p.x == x && p.y == y;
}
```

- ***Liskov substitution principle*** any important property of a type should also hold for its subtype.
  - Any method written for the type should work equally well on its subtypes.

Completion of prior example

- Now consider a different subclass `CounterPoint`
  - Question: What happens to clients of `Point`?
  - Answer: `CounterPoint` objects behave badly ☹

```
public class CounterPoint extends Point
    private static final AtomicInteger counter =
       new AtomicInteger();

    public CounterPoint(int x, int y) {
       super (x, y);
       counter.incrementAndGet();
    }
    public int numberCreated() { return counter.get(); }
 }
```

The real lesson

- There is **no way** to extend an *instantiable* class and add an aspect while preserving the equals contract.
- Wow! Inheritance is hard!
- Workaround: Favor composition over inheritance (Item 16). ColorPoint.java
- Note: This was not well understood when some Java libraries were built…
  - java.sql.Timestamp extends java.util.Date (adding a *nanoseconds* field)
  - Implementation of Timestamp.equals violates symmetry

How to implement equals()

1) Use == to see if argument is a reference to this (optimization)
2) Use instanceof to check if argument is of the correct type (properly handles null)
3) Cast the argument to the correct type
4) Check each "significant" field
   - See next slide
5) Check reflexivity, symmetry, transitivity

Common practices for comparing

- Use __ for primitive fields other than float or double
- Invoke __ method recursively for object reference fields
- Use __ for float fields and __ for double fields
- Use __ for array fields (jdk 1.5+ only)
- For object reference fields may legitimately contain null, use one of the following idioms to avoid NPE:

Common practices for comparing

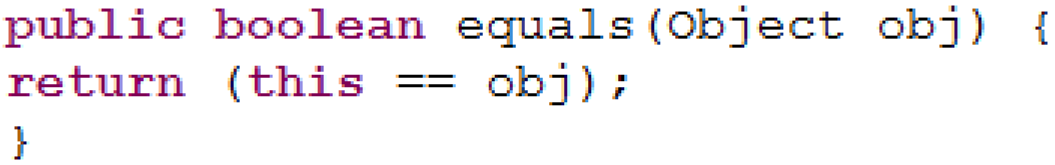
- Use *==* for primitive fields other than float or double
- Invoke *equals* method recursively for object reference fields
- Use *Float.compare* for float fields and *Double.compare* for double fields
- Use *Arrays.equals* for array fields (jdk 1.5+ only)
- For object reference fields may legitimately contain null, use one of the following idioms to avoid NPE:
  - *(field == null ? o.field == null : field.equals (o.field))*
  - *(field == o.field || (field != null && field.equals (o.field)))*

What not to do

- Don't be too clever
- Don't substitute another type for Object
  - `@Override`

    ```
    public boolean equals (MyClass o)
    ```
    - Wrong, but `@Override` tag guarantees compiler will catch problem
  - Overloads `equals()` – does not override it!
- Don't throw `NullPointerException` or `ClassCastException`

Default implementation of equals

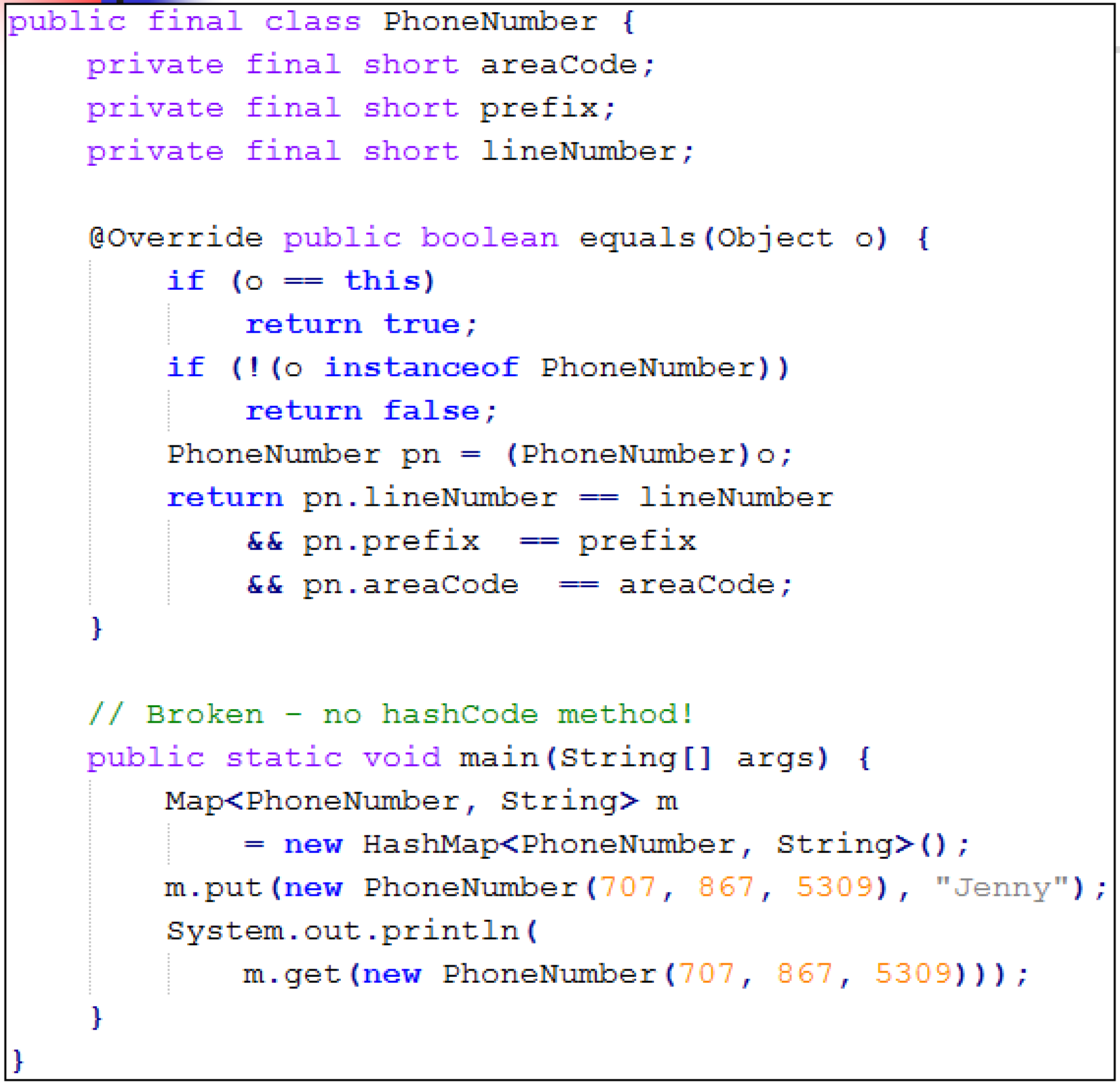


```
public boolean equals(Object obj)  {
return (this == obj);
}
```

Item 11 Always override *hashCode* when you override *equals*

- Always override `hashCode()` when you override `equals()`
- Contract:
  1) `hashCode()` must return same integer on multiple calls, as long as `equals()` unchanged
  2) If `x.equals(y)`, then x, y have same hashcode
  3) It is **not** required that unequal objects have different hashcodes.

Code Example



```java
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNumber == lineNumber
            && pn.prefix  == prefix
            && pn.areaCode  == areaCode;
    }

    // Broken - no hashCode method!
    public static void main(String[] args) {
        Map<PhoneNumber, String> m
            = new HashMap<PhoneNumber, String>();
        m.put(new PhoneNumber(707, 867, 5309), "Jenny");
        System.out.println(
            m.get(new PhoneNumber(707, 867, 5309)));
    }
}
```

Demo

Second provision is key

- Suppose `x.equals(y)`, but x and y have different values for `hashCode()`
- Consider this code:

```
Map m = new HashMap();
m.put(x, "Hello");  // expect x to map to Hello
// m.get(y) should return Hello,
// since x.equals(y), but it doesn't!
```

- HashMap/HashSet/Hashtable will not function properly…

How to implement hashCode

- Avoid really bad implementations
  - `@Override public int hashCode() { return 42;}`
  - Hash table now performs terribly (but, at least, correctly…)
- Start with some nonzero value (e.g 17)
- (Repeatedly) compute int hashCode "c" for each "significant field"
  - Various rules for each data type
- Combine: `result = result*37 + c;`
- `Detailed Steps(EF Item 11)`

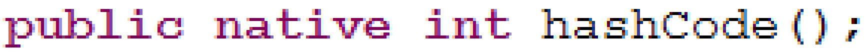
Optimize hashCode() for immutable objects

- No reason to recompute hashcode

```
// Lazy initialization example
 private int hashCode = 0;
 @Override public int hashCode() {
    if (hashCode == 0)
       { … }  // needed now, so compute hashCode
    else return hashCode;
 }
```

Default implementation of hashCode

- As much as is reasonably practical, the hashCode method defined by class Object does return *distinct integers for distinct objects*.
  - This is typically implemented by converting the ***internal address*** of the object into an integer, but this implementation technique is not required by the Java$^{\text{TM}}$ programming language.

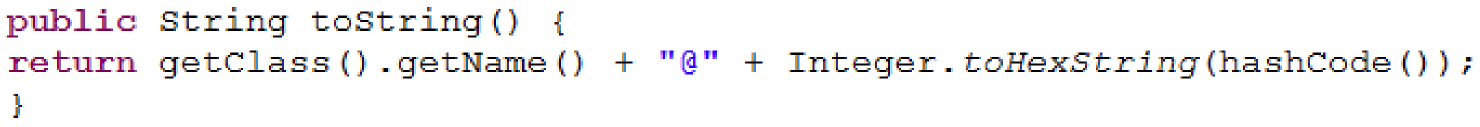


```
public native int hashCode();
```

Item 12 Always override *toString*

- Default Implementation
  - The *toString* method for class Object returns a string like

    class name + “@” + unsigned hexadecimal representation of the *hash code* of the object



```
public String toString() {
return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

- Always override `toString()`

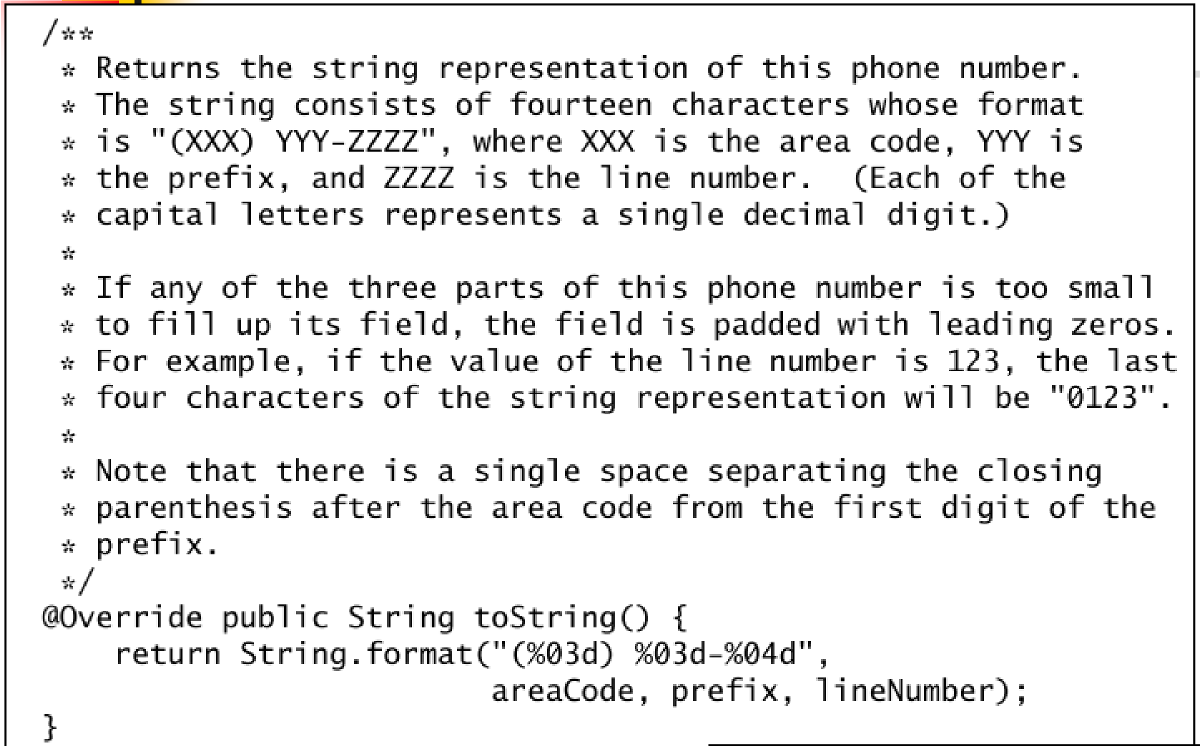
Content of toString

- Return all the "interesting" information in an object
  - Helpful to provide diagnostic message:
    - System.out.println("Failed to connect: " + phoneNumber);
    - "{Jenny=(707867-5309)}" is more pleasant than "{Jenny=PhoneNumber@163b91}"
- Return a summary if the object is large or contains state that is not conducive (有助的) to string representation.
  - e.g. "Manhattan white pages (1487536 listings)" or "Thread[main,5,main]"

About the format of the return value

- Should the format of the return value be specified in the documentation?
  - Not specified → return value only for human reading
  - Specified → both human & program use it
    - May parse the representation
    - Very hard to change the representation in a future release.
- Whether or not,
  - Always document your intentions clearly
    - See next slide for code examples.
  - And provide getters for values toString() provides
    - Do not force clients to parse String representation

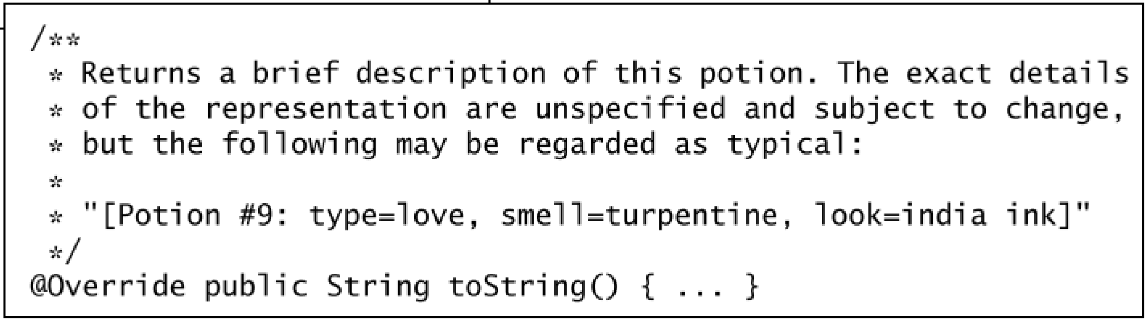
Code examples



```
/**
 * Returns the string representation of this phone number.
 * The string consists of fourteen characters whose format
 * is "(XXX) YYY-ZZZZ", where XXX is the area code, YYY is
 * the prefix, and ZZZZ is the line number.  (Each of the
 * capital letters represents a single decimal digit.)
 *
 * If any of the three parts of this phone number is too small
 * to fill up its field, the field is padded with leading zeros.
 * For example, if the value of the line number is 123, the last
 * four characters of the string representation will be "0123".
 *
 * Note that there is a single space separating the closing
 * parenthesis after the area code from the first digit of the
 * prefix.
 */
@Override public String toString() {
    return String.format("(%03d) %03d-%04d",
                         areaCode, prefix, lineNumber);
}
```

← Format specified

Format not specified →



```
/**
 * Returns a brief description of this potion. The exact details
 * of the representation are unspecified and subject to change,
 * but the following may be regarded as typical:
 *
 * "[Potion #9: type=love, smell=turpentine, look=india ink]"
 */
@Override public String toString() { ... }
```

Item 13 Override clone() judiciously

- Override `clone()` judiciously
- Cloneable is a *"mixin" interface*
  - Unfortunately, it fails to provide any methods
    - `clone()` is defined in `Object` (protected)
- Mixin interface:

  A mixin is a type that a class can implement in addition to its "primary type" to declare that it provides some optional behavior.

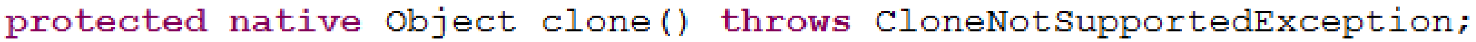
  For example, Comparable is a mixin interface that allows a class to declare that its instances are ordered with respect to other mutually comparable objects. Such an interface is called a mixin because it allows the optional functionality to be "mixed in" to the type's primary functionality.

The Contract is weak

- Contract :
  - Create a copy such that `x.clone() != x`
  - `x.clone().getClass() == x.getClass()`
  - Should have `x.clone().equals(x)`
  - No constructors are called

Default implementation of clone

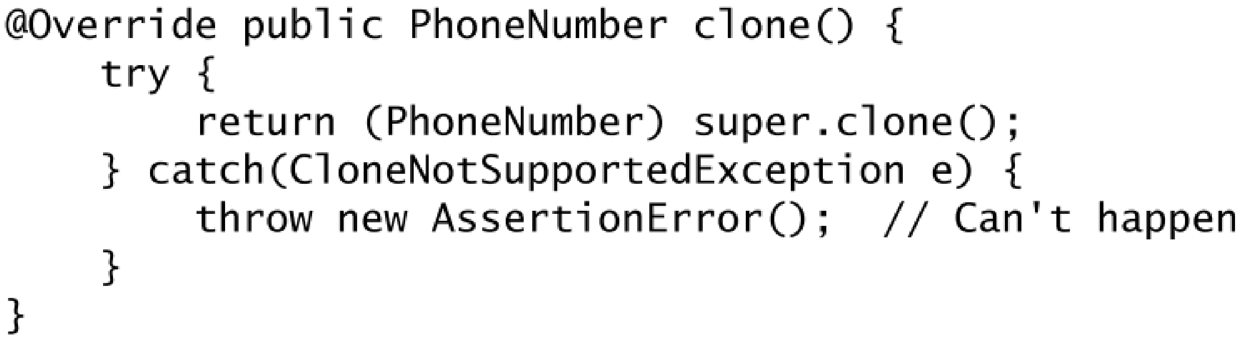
- The method clone() for class Object performs a specific cloning operation.
  - First, *<u>CloneNotSupportedException is thrown</u>* if the class of this object does not implement the interface Cloneable.
    - Note: arrays are considered to implement the interface Cloneable.
  - Otherwise,
    - a new instance of the class of this object is created
    - and all its fields are initialized with exactly the contents of the corresponding fields of this object, as if by assignment; the contents of the fields are not themselves cloned.
  - Thus, this method performs a "***shallow copy***" of this object, not a "deep copy" operation.



```
protected native Object clone() throws CloneNotSupportedException;
```

The simplest case

- If every fields of a class contains only *primitive type or reference to immutable object*, override the *clone* method is easy.
- Two steps
  1) Override clone() with a ***public*** method whose *return type is the class itself*
  2) This method call *super.clone*



```
@Override public PhoneNumber clone() {
    try {
        return (PhoneNumber) super.clone();
    } catch(CloneNotSupportedException e) {
        throw new AssertionError();  // Can't happen
    }
}
```

The role of mutability

- If a class has only primitive fields or immutable references as fields, `super.clone()` returns exactly what you want.
- For objects with mutable references, "deep copies" are required.
- Example: cloning a `Stack` class that uses an `Array` for a representation.
  - Representation `Array` must also be cloned.
  - So, call `super.clone()`, then clone `Array`

Other Cloning problems

- Cloning may be a problem with *final fields*
- Cloning recursively may not be sufficient
  - See HashTable example in the book
- Result:
  - You may be better off **NOT** implementing Cloneable (even never invoking, except array copying)
  - Providing a separate copy mechanism may be preferable.

Alternatives

- Copy Constructor
  - `public Yum (Yum yum)`
- Copy Factory
  - `public static Yum newInstance(Yum yum)`
- Advantages
  - Don’t rely only a risk-prone extralinguistic(语言之外) object creation mechanism.
  - Don’t demand unenforceable adherence to thinly documented conventions
  - Don’t conflict with the proper use of final fields.
  - Don’t throw unnecessary checked exception.
  - Don’t require casts.

Item 14 Consider implementing Comparable

- Consider Implementing Comparable
- Contract
  1) Returns negative, zero, or positive depending on order of this and specified object
  2) `sgn(x.compareTo(y) == -sgn(y.compareTo(x))`
  3) `compareTo()` must be transitive
  4) If `x.compareTo(y) == 0`, x and y must consistently compare to all values z.
  5) <u>Recommended</u> that `x.compareTo(y) == 0` iff `x.equals(y)`
  6) Note that `compareTo()` can throw *ClassCastExceptions*

Elements of the contract

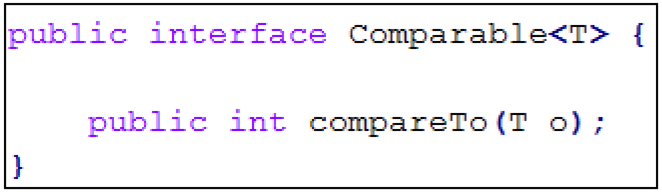
- The same issue with `equals()` arises in the case of inheritance:
  - There is simply no way to extend an instantiable class with a new aspect while preserving the `compareTo()` contract.
  - Same workaround – Favor composition over inheritance
- Some Java classes violate *the consistency requirement with `equals()`*.

BigDecimal Example

- Example: The `BigDecimal` class

```
//This is horrible!
  Object x = new BigDecimal(“1.0”);
  Object y = new BigDecimal(“1.00”);
// !x.equals(y), but x.compareTo(y) == 0

  Set s = new HashSet();
  Set t = new TreeSet();

  s.add(x); s.add(y);
// HashSet uses equals, so s has 2 elements

  t.add(x); t.add(y);
// TreeSet uses compareTo, so t has 1 element
```

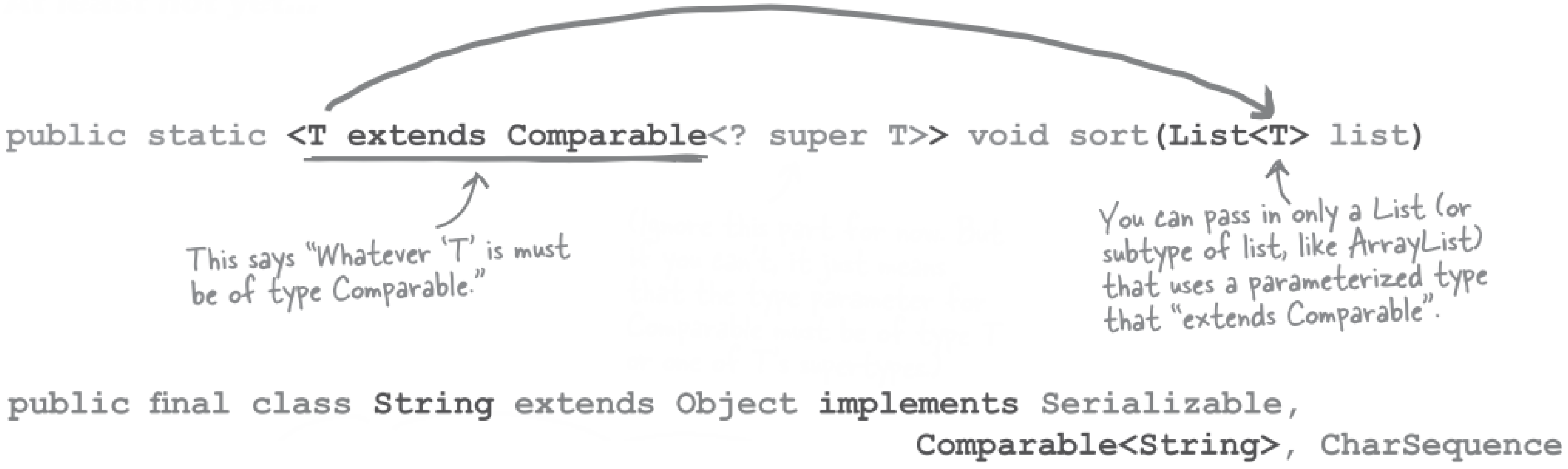
Comparable & Generic



```
public interface Comparable<T> {

    public int compareTo(T o);
}
```

← The Comparable interface

Use of the Comparable interface
(in *Collections.java*)



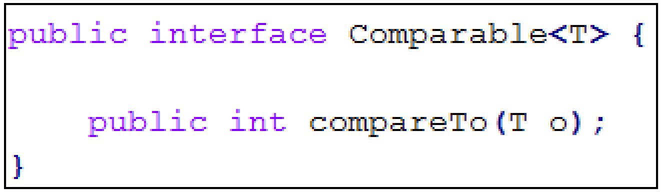
```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

This says “Whatever ‘T’ is must be of type Comparable.”

You can pass in only a List (or subtype of list, like ArrayList) that uses a parameterized type that “extends Comparable”.

```
public final class String extends Object implements Serializable,
                                             Comparable<String>, CharSequence
```

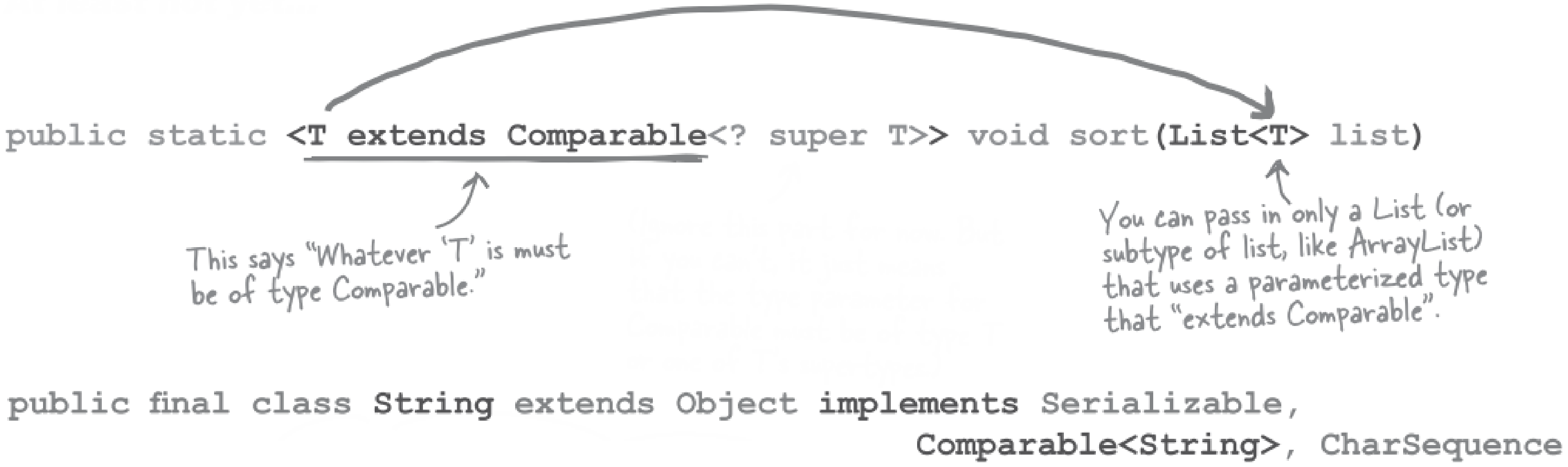
Comparable & Generic



```
public interface Comparable<T> {

    public int compareTo(T o);
}
```

← The Comparable interface

Use of the Comparable interface
(in *Collections.java*)



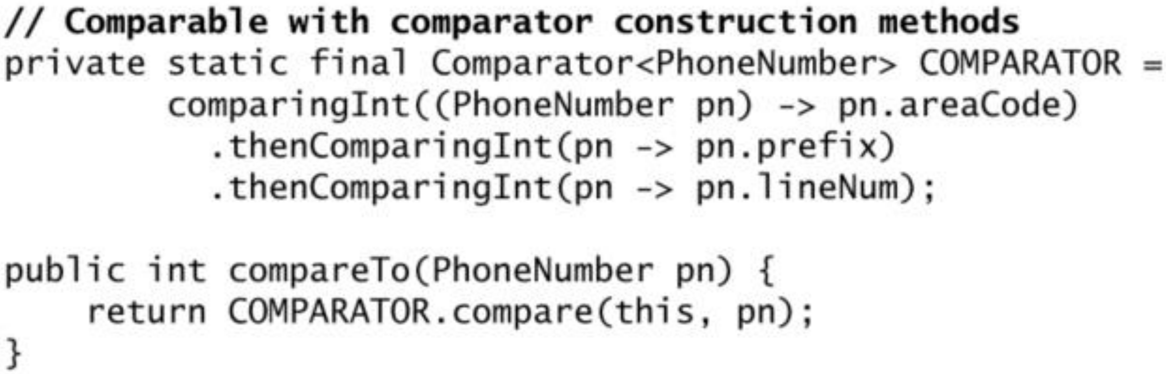
```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

This says "Whatever 'T' is must be of type Comparable."

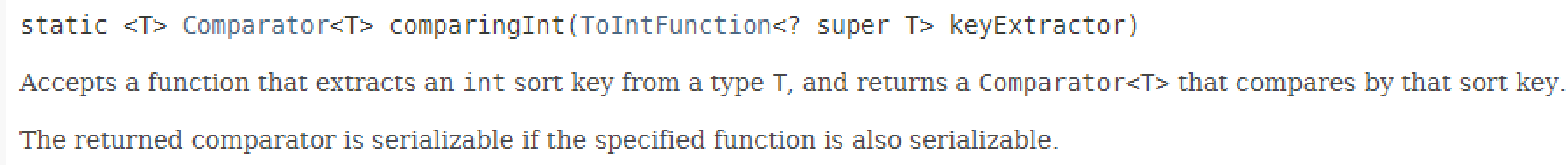
You can pass in only a List (or subtype of list, like ArrayList) that uses a parameterized type that "extends Comparable".

```
public final class String extends Object implements Serializable,
                                             Comparable<String>, CharSequence
```

Comparator construction methods (比较器构造方法)



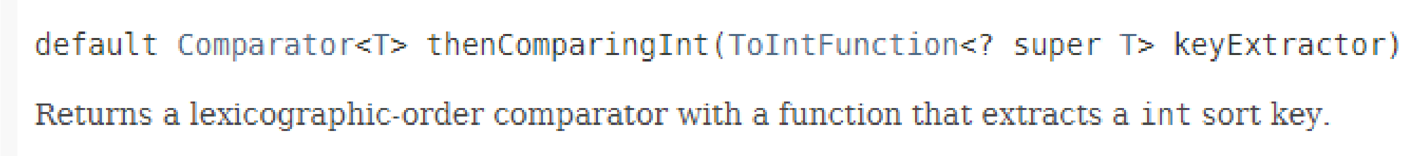
```
// Comparable with comparator construction methods
private static final Comparator<PhoneNumber> COMPARATOR =
        comparingInt((PhoneNumber pn) -> pn.areaCode)
          .thenComparingInt(pn -> pn.prefix)
          .thenComparingInt(pn -> pn.lineNum);

public int compareTo(PhoneNumber pn) {
    return COMPARATOR.compare(this, pn);
}
```



`static <T> Comparator<T> comparingInt(ToIntFunction<? super T> keyExtractor)`

Accepts a function that extracts an `int` sort key from a type `T`, and returns a `Comparator<T>` that compares by that sort key.

The returned comparator is serializable if the specified function is also serializable.



`default Comparator<T> thenComparingInt(ToIntFunction<? super T> keyExtractor)`

Returns a lexicographic-order comparator with a function that extracts a `int` sort key.