

# Principles of Software Construction: Objects, Design, and Concurrency

## Testing and Object Methods in Java

**Josh Bloch**

Charlie Garrod

Darya Melicher



# Administrivia

- Homework 1 due **Today** 11:59 p.m.
  - Everyone must read and sign our collaboration policy
  - TAs will be available to help you
  - You have late days, but you might want to save for later
- Second homework will be posted shortly

# Key concepts from Tuesday

- Interfaces-based designs are flexible
- Information hiding is crucial to good design
- Exceptions are way better than error codes

# Unfinished Business: Exceptions

# Creating and throwing your own exceptions

```
public class SpanishInquisitionException extends RuntimeException {  
    public SpanishInquisitionException() {  
    }  
}  
  
public class HolyGrail {  
    public void seek() {  
        ...  
        if (heresyByWord() || heresyByDeed())  
            throw new SpanishInquisitionException();  
        ...  
    }  
}
```

# Benefits of exceptions

- You can't forget to handle common failure modes
  - Compare: using a flag or special return value
- Provide high-level summary of error, and stack trace
  - Compare: core dump in C
- Improve code structure
  - Separate normal code path from exceptional
  - Ease task of recovering from failure
- Ease task of writing robust, maintainable code

# Guidelines for using exceptions (1)

- Avoid unnecessary checked exceptions (EJ Item 71)
- Favor standard exceptions (EJ Item 72)
  - `IllegalArgumentException` – invalid parameter value
  - `IllegalStateException` – invalid object state
  - `NullPointerException` – null param where prohibited
  - `IndexOutOfBoundsException` – invalid index param
- Throw exceptions appropriate to abstraction (EJ Item 73)

# Guidelines for using exceptions (2)

- Document all exceptions thrown by each method
  - Checked and unchecked (EJ Item 74)
  - But don't *declare* unchecked exceptions!
- Include failure-capture info in detail message (Item 75)
  - `throw new IllegalArgumentException("Modulus must be prime: " + modulus);`
- Don't ignore exceptions (EJ Item 77)
  - // Empty catch block IGNORES exception - Bad smell in code!**
  - ```
try {  
    ...  
} catch (SomeException e) { }
```



# Remember this slide?

*You can do much better!*

```
FileInputStream fileInput = null;
try {
    FileInputStream fileInput = new FileInputStream(fileName);
    DataInput dataInput = new DataInputStream(fileInput);
    return dataInput.readInt();
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " + fileName);
} catch (IOException e) {
    System.out.println("Couldn't read file: " + e);
} finally {
    if (fileInput != null) fileInput.close();
}
```

# Manual resource termination is ugly and error prone

- Even good programmers usually get it wrong
  - Sun’s Guide to Persistent Connections got it wrong in code that claimed to be exemplary
  - Solution on page 88 of Bloch and Gafter’s *Java Puzzlers* is badly broken; no one noticed for years
- 70% of the uses of the `close` method in the JDK itself were wrong in 2008(!)
- Even “correct” idioms for manual resource management are deficient

# The solution: try-with-resources (TWR)

## *Automatically closes resources*

```
try (DataInput dataInput =  
    new DataInputStream(new FileInputStream(fileName))) {  
    return dataInput.readInt();  
} catch (FileNotFoundException e) {  
    System.out.println("Could not open file " + fileName);  
} catch (IOException e) {  
    System.out.println("Couldn't read file: " + e);  
}
```

# File copy without TWR

```
static void copy(String src, String dest) throws IOException {  
    InputStream in = new FileInputStream(src);  
    try {  
        OutputStream out = new FileOutputStream(dest);  
        try {  
            byte[] buf = new byte[8 * 1024];  
            int n;  
            while ((n = in.read(buf)) >= 0)  
                out.write(buf, 0, n);  
            } finally {  
                out.close();  
            }  
        } finally {  
            in.close();  
        }  
    }  
}
```

# File copy with TWR

```
static void copy(String src, String dest) throws IOException {  
    try (InputStream in = new FileInputStream(src);  
        OutputStream out = new FileOutputStream(dest)) {  
        byte[] buf = new byte[8 * 1024];  
        int n;  
        while ((n = in.read(buf)) >= 0)  
            out.write(buf, 0, n);  
    }  
}
```

# Outline

- I. Specifying program behavior – contracts
- II. Testing correctness – Junit and friends
- III. Overriding Object methods

# What is a contract?

- Agreement between an object and its user
- Includes
  - Method signature (type specifications)
  - Functionality and correctness expectations
  - Performance expectations
- **What the method does**, not how it does it
  - Interface (**API**), not implementation

# Method contract details

- States method's and caller's responsibilities
- Analogy: legal contract
  - If you pay me this amount on this schedule...
  - I will build a with the following detailed specification
  - Some contracts have remedies for nonperformance
- Method contract structure
  - **Preconditions:** what method requires for correct operation
  - **Postconditions:** what method establishes on completion
  - **Exceptional behavior:** what it does if precondition violated
- Defines what it means for impl to be correct



# Formal contract specification

## *Java Modelling Language (JML)*

```
/*@ requires len >= 0 && array != null && array.length == len;  
  @  
  @ ensures \result ==  
    @      (\sum int j; 0 <= j && j < len; array[j]);  
  @*/  
int total(int array[], int len);
```



precondition



postcondition

- Theoretical approach

- Advantages

- Runtime checks generated automatically
    - Basis for formal verification
    - Automatic analysis tools

- Disadvantages

- Requires a lot of work
    - Impractical in the large
    - Some aspects of behavior not amenable to formal specification

# Textual specification - Javadoc

- Practical approach
- Document
  - Every parameter
  - Return value
  - Every exception (checked and unchecked)
  - What the method does, including
    - Purpose
    - Side effects
    - Any thread safety issues
    - Any performance issues
- Do **not** document implementation details

# Specifications in the real world

## *Javadoc*

```
/**
 * Returns the element at the specified position of this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant time.
 * In some implementations, it may run in time proportional to the
 * element position.
 *
 * @param index position of element to return; must be non-negative and
 *         less than the size of this list.
 * @return the element at the specified position of this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```



postcondition



precondition

(No side effects)

# Outline

- I. Specifying program behavior – contracts
- II. Testing correctness – Junit and friends
- III. Overriding Object methods

# Semantic correctness

## *adherence to contracts*

- Compiler ensures types are correct (type-checking)
  - Prevents many runtime errors, such as “Method Not Found” and “Cannot add boolean to int”
- Static analysis tools (e.g., FindBugs) recognize many common problems (*bug patterns*)
  - Overriding equals without overriding hashCode
- But how do you ensure semantic correctness?

# Formal verification

- Use mathematical methods to prove correctness with respect to the formal specification
- Formally prove that all possible executions of an implementation fulfill the specification
- Manual effort; partial automation; not automatically decidable

**"Testing shows the presence,  
not the absence of bugs."**

Edsger W. Dijkstra, 1969

# Testing

- Executing the program with selected inputs in a controlled environment
- Goals
  - Reveal bugs, so they can be fixed (main goal)
  - Assess quality
  - Clarify the specification, documentation

**“Beware of bugs in the above code; I  
have only proved it correct, not tried it.”**  
Donald Knuth, 1977

# Who's right, Dijkstra or Knuth?

- They're both right!
- **Please see “Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken”**
  - Official “Google Research” blog
  - <http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html>
- There is no silver bullet
  - Use all tools at your disposal



# Manual testing?

GENERIC TEST CASE: USER SENDS MMS WITH PICTURE ATTACHED.

| Step ID | User Action                 | System Response             |
|---------|-----------------------------|-----------------------------|
| 1       | Go to Main Menu             | Main Menu appears           |
| 2       | Go to Messages Menu         | Message Menu appears        |
| 3       | Select "Create new Message" | Message Editor screen opens |
| 4       | Add Recipient               | Recipient is added          |
| 5       | Select "Insert Picture"     | Insert Picture Menu opens   |
| 6       | Select Picture              | Picture is Selected         |
| 7       | Select "Send Message"       | Message is correctly sent   |

- Live System?
- Extra Testing Sy
- Check output / assertions?
- Effort, Costs?
- Reproducible?



# Automate testing

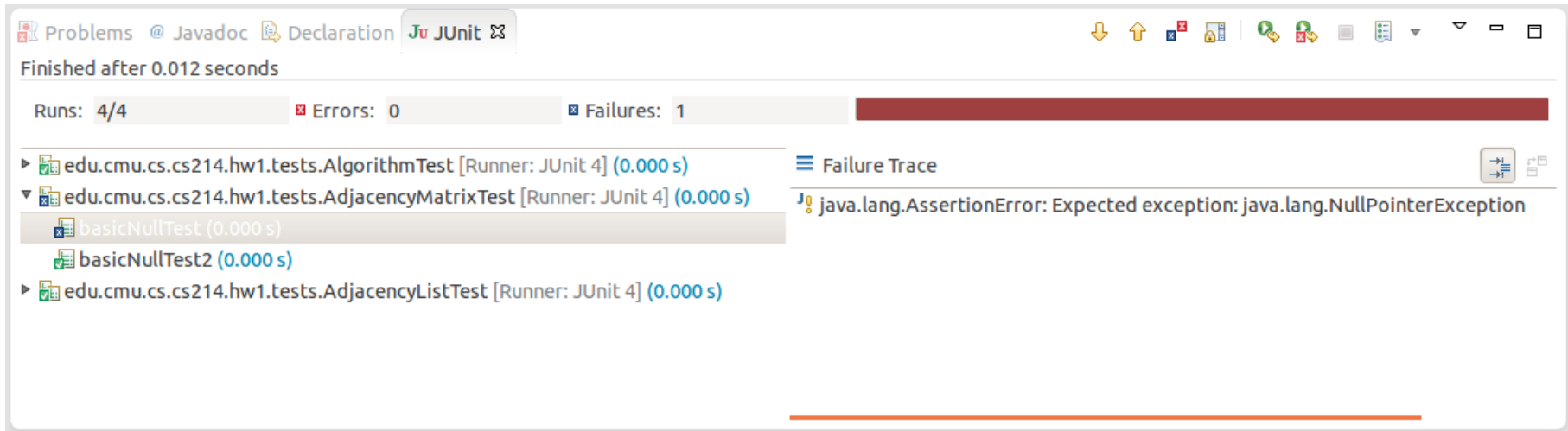
- Execute a program with specific inputs, check output for expected values
- Set up testing infrastructure
- **Execute tests regularly**
  - After *every* change

# Unit tests

- Unit tests for small units: methods, classes, subsystems
  - Smallest testable part of a system
  - Test parts before assembling them
  - Intended to catch local bugs
- Typically written by developers
- Many small, fast-running, independent tests
- Few dependencies on other system parts or environment
- Insufficient, but a good starting point

# JUnit

- Popular unit-testing framework for Java
- Easy to use
- Tool support available
- Can be used as design mechanism



# Kent Beck on automated testing

*“Functionality that can’t be demonstrated by automated test simply don't exist.”*

# Selecting test cases: common strategies

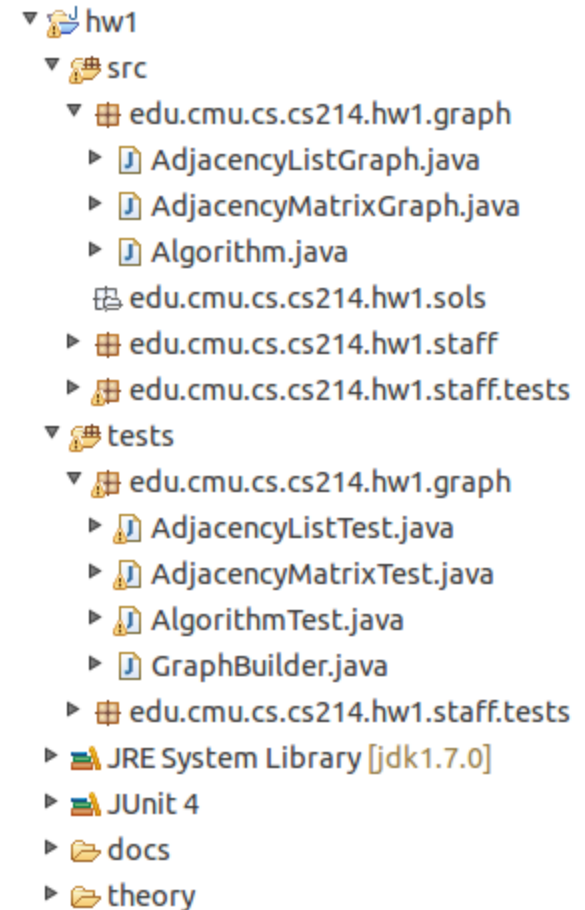
- Read specification
- Write tests for
  - Representative case
  - Invalid cases
  - Boundary conditions
- Write stress tests
  - Automatically generate huge numbers of test cases
- Think like an attacker
  - The tester's goal is to find bugs!
- How many test should you write?
  - Aim to cover the specification
  - Work within time/money constraints

# JUnit conventions

- TestCase collects multiple tests (in one class)
- TestSuite collects test cases (typically package)
- Tests should run fast
- Tests should be independent
- Tests are methods without parameter and return value
- AssertionError signals failed test (unchecked exception)
- Test Runner knows how to run JUnit tests
  - (uses reflection to find all methods with @Test annotat.)

# Test organization

- Conventions (not requirements)
- Have a test class FooTest for each public class Foo
- Have a source directory and a test directory
  - Store FooTest and Foo in the same package
  - Tests can access members with default (package) visibility





# Testable code

- **Think about testing when writing code**
- Unit testing encourages you to write testable code
- Modularity and testability go hand in hand
- Same test can be used on multiple implementations of an interface!
- Test-Driven Development
  - A design and development method in which you **write tests before you write the code**
  - Writing tests can expose API weaknesses!

# Run tests frequently

- You should only commit code that is passing all tests
- Run tests before every commit
- If entire test suite becomes too large and slow for rapid feedback:
  - Run local package-level tests ("smoke tests") frequently
  - Run all tests nightly
  - Medium sized projects easily have 1000s of test cases
- Continuous integration servers help to scale testing

# Continuous integration - Travis CI

The screenshot displays the Travis CI web interface for the repository `wyvernlang / wyvern`. The build status is `passing`. The interface includes a sidebar with a search bar and a list of repositories, showing `wyvernlang/wyvern` with build #17. The main content area shows details for `Build #17`, including a green checkmark icon, the commit message `SimpleWyvern-devel Asserting false (works on Linux, so its OK).`, the commit hash `fd7be1c`, and the duration `ran for 16 sec`. A yellow banner indicates that the job ran on legacy infrastructure. Below this, a log viewer shows the build steps, including cloning the repository, switching to Oracle JDK8, and running the `ant test` command.

Build #17 - wyvernlang / wyvern

Travis CI Blog Status Help

Jonathan Aldrich

Search all repositories

My Repositories +

- ✓ wyvernlang/wyvern #17
  - Duration: 16 sec
  - Finished: 3 days ago

wyvernlang / wyvern build passing

Current Branches Build History Pull Requests > Build #17 Settings

✓ SimpleWyvern-devel Asserting false (works on Linux, so its OK). # 17 passed

- Commit fd7be1c
- Compare 0e2af1f..fd7be1c
- ran for 16 sec
- 3 days ago

potanin authored and committed

This job ran on our legacy infrastructure. Please read [our docs on how to upgrade](#)

Remove Log Download Log

```
1 Using worker: worker-linux-027f0490-1.bb.travis-ci.org:travis-linux-2
2
3 Build system information system_info
67
68 $ git clone --depth=50 --branch=SimpleWyvern-devel git.checkout 0.815
69 $ jdk_switcher use oraclejdk8
70 Switching to Oracle JDK8 (java-8-oracle), JAVA_HOME will be set to /usr/lib/jvm/java-8-oracle
71 $ java -Xmx32m -version
72 java version "1.8.0_31"
73 Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
74 Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)
75 $ javac -J-Xmx32m -version
76 javac 1.8.0_31
77 $ cd tools 0.015
78
79 The command "cd tools" exited with 0.
80 $ ant test 11.815
81 Buildfile: /home/travis/build/wyvernlang/wyvern/tools/build.xml
82
83 copper-compose-compile:
```

Automatically builds, tests, and displays the result

# Continuous integration - Travis CI

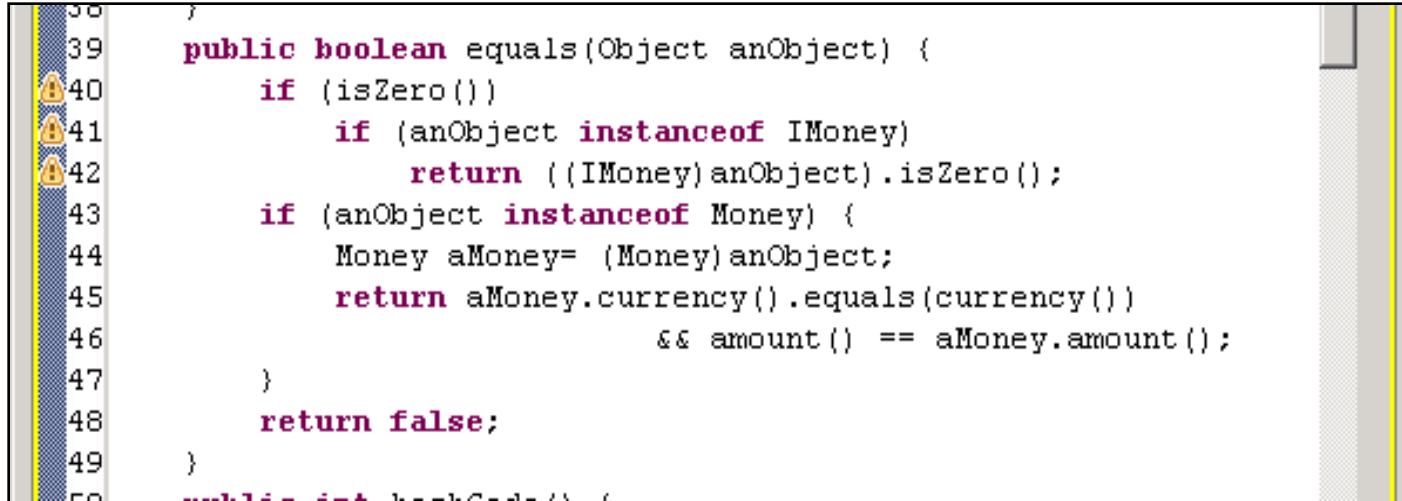
Travis CI Build History for wyvernlang / wyvern

| Build Status | Commit Message                                  | Commit ID | Duration | Time Ago    |
|--------------|-------------------------------------------------|-----------|----------|-------------|
| ✓            | SimpleWyvern-devel Asserting false (works on L  | fd7be1c   | 16 sec   | 3 days ago  |
| ✓            | SimpleWyvern-devel Debugging mac bug.           | 0e2af1f   | 22 sec   | 3 days ago  |
| ✓            | SimpleWyvern-devel Zooming in on Mac's IRBui    | 8b3606f   | 15 sec   | 4 days ago  |
| ✓            | SimpleWyvern-devel Zooming in on Mac LLVM b     | 727fc84   | 16 sec   | 4 days ago  |
| ✓            | SimpleWyvern-devel Removed outdated tests       | 4684fb5   | 15 sec   | 11 days ago |
| ✓            | newlexer Merge branch 'master' of https://githu | 876a074   | 14 sec   | 11 days ago |
| ✓            | master Build with JDK 8                         | b15273c   | 13 sec   | 11 days ago |
| ✗            | master fixed Travis build script syntax error   | 737a89f   | 5 sec    | 11 days ago |

You can see the results of builds over time

# Outlook: statement coverage

- Trying to test all parts of the implementation
- Execute every statement, ideally



```
38     }
39     public boolean equals(Object anObject) {
40         if (isZero())
41             if (anObject instanceof IMoney)
42                 return ((IMoney)anObject).isZero();
43         if (anObject instanceof Money) {
44             Money aMoney= (Money)anObject;
45             return aMoney.currency().equals(currency())
46                     && amount() == aMoney.amount();
47         }
48         return false;
49     }
50     public int hashCode() {
```

- Does 100% coverage guarantee correctness?

# Outline

- I. Specifying program behavior – contracts
- II. Testing correctness – Junit and friends
- III. Overriding `Object` methods

# Methods common to all objects

- How do collections know how to test objects for equality?
- How do they know how to hash and print them?
- The relevant methods are all present on `Object`
  - **`equals`** - returns `true` if the two objects are “equal”
  - **`hashCode`** - returns an `int` that must be equal for equal objects, and is likely to differ on unequal objects
  - **`toString`** - returns a printable string representation

# Object implementations

- Provide *identity semantics*
  - `equals(Object o)` - returns true if o refers to this object
  - `hashCode()` - returns a near-random `int` that never changes over the object lifetime
  - `toString()` - returns a nasty looking string consisting of the type and hash code
    - For example: `java.lang.Object@659e0bfd`



# Overriding Object implementations

- **(nearly) Always override toString**
  - println invokes it automatically
  - Why settle for ugly?
- **No need to override equals and hashCode if you want identity semantics**
  - When in doubt, don't override them
  - It's easy to get it wrong

# Overriding toString

## Overriding toString is easy and beneficial

```
final class PhoneNumber {  
    private final short areaCode;  
    private final short prefix;  
    private final short lineNumber;  
    ...  
    @Override public String toString() {  
        return String.format("(%03d) %03d-%04d",  
                               areaCode, prefix, lineNumber);  
    }  
}
```

```
PhoneNumber jenny = ...;  
System.out.println(jenny);  
Prints: (707) 867-5309
```

# The equals contract

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 true if and only if *y.equals(x)* returns true.
- **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.
- **Consistent**: For any non-null reference values *x* and *y*, multiple invocations of *x.equals(y)* consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value *x*, *x.equals(null)* must return false.

# The equals contract in English

- **Reflexive** – every object is equal to itself
- **Symmetric** – if `a.equals(b)` then `b.equals(a)`
- **Transitive** – if `a.equals(b)` and `b.equals(c)`, then `a.equals(c)`
- **Consistent** – equal objects stay equal unless mutated
- **“Non-null”** – `a.equals(null)` returns false
- Taken together these ensure that equals is a global **equivalence relation** over all objects

# equals Override Example

```
public final class PhoneNumber {  
    private final short areaCode;  
    private final short prefix;  
    private final short lineNumber;  
  
    @Override public boolean equals(Object o) {  
        if (!(o instanceof PhoneNumber)) // Does null check  
            return false;  
        PhoneNumber pn = (PhoneNumber) o;  
        return pn.lineNumber == lineNumber  
            && pn.prefix == prefix  
            && pn.areaCode == areaCode;  
    }  
  
    ...  
}
```

# The hashCode contract

Whenever it is invoked on the same object more than once during an execution of an 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(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 hash tables.

# The hashCode contract in English

- Equal objects **must** have equal hash codes
  - If you override equals you must override hashCode
- Unequal objects **should** have different hash codes
  - Take all value fields into account when constructing it
- Hash code must not change unless object mutated

# hashCode override example

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

    @Override public int hashCode() {
        int result = 17; // Nonzero is good
        result = 31 * result + areaCode; // Constant must be odd
        result = 31 * result + prefix;   // " " " "
        result = 31 * result + lineNumber; // " " " "
        return result;
    }

    ...
}
```



# Alternative hashCode override

*Less efficient, but otherwise equally good!*

```
public final class PhoneNumber {  
    private final short areaCode;  
    private final short prefix;  
    private final short lineNumber;  
  
    @Override public int hashCode() {  
        return Objects.hash(areaCode, prefix, lineNumber);  
    }  
  
    ...  
}
```

**A one liner. No excuse for failing to override hashCode!**

**For more than you want to know about overriding object methods, see *Effective Java* Chapter 2**

# The == operator vs. equals method

- For primitives you *must* use ==
- For object reference types
  - The == operator provides *identity semantics*
    - Exactly as implemented by Object.equals
    - Even if Object.equals has been overridden
    - This is seldom what you want!
  - **You should (almost) always use .equals**
  - Using == on an object reference is a **bad smell in code**  

```
if (input == "yes")    // A bug!!!
```

# Summary

- Use try-with-resources, not manual cleanup
- Contracts specify correct method behavior
  - Document contract of every method
- Test early, test often!
- Always override toString
- Override equals when you need value semantics
- Override hashCode when your override equals