Development Approach

Principles of Application Development

- Incremental
- Task-oriented
- Test-first
- Continuous integration

Incremental

- Classes, subsystems, systems are designed, coded, tested incrementally
- At each level, compilable, tested, runnable code is archived

Task-Oriented

- “Task-oriented” means user tasks
- “Chunked” into units of deliverable functionality

Test-First Development

- “You’re not ready to write code until you have a test that fails.”
- Create class and method stubs
- Create or generate test stubs
- Add test code to stubs so test fails
- Write method code to pass test

Continuous Integration

- New components are continuously integrated into the application
- Write the top-level application stubs
- Write the top-level application tests
- Write and test stub implementations
- Integrate new code and test the top-level application

JUnit

Ant Integration

- **Run a single test case**

```
<target name="test" depends="jar"
      description="Run selected JUnit test">
  <junit fork="true" >
    <formatter type="plain" usefile="false"/>
    <classpath>
      <path refid="path.class"/>
    </classpath>
    <test fork="yes" name="${testclass}"/>
  </junit>
</target>
```

```
> ant -Dtestclass=mypackage.MyTest test
```

Ant Integration

- **Run a batch of test cases**

```
<target name="test" depends ="jar"
        description="Run the JUnit tests">
  <junit fork="true" haltonfailure="yes">
    <formatter type="plain" usefile="false"/>
    <formatter type="xml" usefile="true"/>
    <classpath>
      <path refid="path.class"/>
    </classpath>
    <batchtest fork="yes" todir="${dir.reports}">
      <fileset dir="${dir.classes}">
        <include name="**/*Test.class"/>
      </fileset>
    </batchtest>
  </junit>
</target>
```

```
> ant test
```

Ant Integration

```
test:
    test:
    [junit] Testsuite: test.RomanNumeralTest
    [junit] Tests run: 1, Failures: 0, Errors: 1, Time elapsed: 0 sec

    [junit]     Caused an ERROR
    [junit] test.RomanNumeralTest
    [junit] java.lang.ClassNotFoundException: test.RomanNumeralTest
    [junit]     at java.net.URLClassLoader$1.run(URLClassLoader.java:200)
    [junit]     at java.security.AccessController.doPrivileged(Native Method)
    [junit]     at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
    [junit]     at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
    [junit]     at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:268)
    [junit]     at java.lang.ClassLoader.loadClass(ClassLoader.java:251)
    [junit]     at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:319)
    [junit]     at java.lang.Class.forName0(Native Method)
    [junit]     at java.lang.Class.forName(Class.java:164)


BUILD FAILED
```

Date

Java Dates

- `Date` **class from Java 1.0 has been superceded by the** `java.util.Calendar` **class**
- `Calendar` **is a factory class**
- **Can't instantiate a** `Calendar` **directly-- use static method** `getInstance()`
  - **This is an example of a Factory Method design pattern**
- **Why? There are different calendars for different locales.**

Creating a Calendar and Getting the Date

```
public static void main(String[] args)
{
   Calendar calendar = Calendar.getInstance();
   SimpleDateFormat dateFormatter =
   newSimpleDateFormat();
   System.out.println(
         dateFormatter.format(calendar.getTime
  ()));
}
```

Dates (cont'd)

- `getInstance()` **with no arguments returns a** `Calendar` **of the current date and time**
- **method** `setTime(Date d)` **allows you to set a specific date**
- **All of the components of the date and time can be set with static methods in the** `Calendar` **class**

Getting the Date You Want

- **Get an instance of** `java.util.Calendar` **(or its subclass** `GregorianCalendar`**)**
  - **This gives you a** `Calendar` **whose date and time represent the instant the** `Calendar` **was instantiated**
- **Use the methods of** `Calendar` **to set the elements of the date or time that you want**
- **Call the** `getTime()` **method to return a** `Date` **object with the date and time you want**

Date "Gotchas"

- Most `Date` methods are deprecated, so you must manipulate a `Calendar` object and then call `getTime()` to do most `Date` manipulation
- When comparing two `Dates`, remember that they contain hours, minutes, and seconds, as well as month, day and year

Formatters

java.util.Formatter

- **Powerful utility for formatting text**
  - **Inspired by C's `printf` function**
- **Primary motivation for introducing variable argument lists in Java 5**
- **Principle method is**

```
public Formatter format(String format,
                        Object ... args)
```

  - **The `format` argument consists of a combination of fixed text and embedded format specifiers**
  - **Implements builder pattern**

Format Specifiers

- **General form:**

  `%[argument_index$][flags][width][.precision]conversion`

  - `argument_index`, **specifies which of the variable args should be used for conversion (only used when a single argument feeds multiple conversions**
  - `flag`, **modifies output, dependent on conversion**
  - `width`, **indicates the minimum number of characters to be written to the output**
  - `precision`, **usually used to restrict the number of characters, specific behavior depends on the conversion**
  - `conversion`, **a character indicating how the argument should be formatted, the set of valid conversions for a given argument depends on the argument's data type**

Conversions

Conversion	Type	Description
'b', 'B'	General	"false" for `null` or false boolean, else "true"
'h', 'H'	General	"null" for `null`, otherwise hash code in hex
's', 'S'	General	"null" for `null`, otherwise per `Formattable`
'c', 'C'	Character	A unicode character
'd'	Integral	A decimal integer
'o'	Integral	An octal integer
'x', 'X'	Integral	A hexadecimal integer
'e', 'E'	Integral	A decimal number in scientific notation
'f'	Floating-point	A decimal number
'g', 'G'	Floating-point	A decimal number may be scientific notation
'a', 'A'	Floating-point	Hexadecimal floating-point
't', 'T'	Date/Time	Prefix for date time conversions
'%'	Percent sign	A literal '%'
'n'	Line separator	Platform specific line separator

Date/Time Conversions

- The ‘t’ or ‘T’ prefix is followed by an additional conversion character for a specific formatting of date/time element
  - Year, month, day
  - Hour, minute, second, …
  - AM/PM

Other Formatters

- **Specialized formatters in** `java.text`
  - `SimpleDateFormat` **formats date/time**
  - `DecimalFormat` **formats decimal numbers**
  - **Use the** `format()` **method to format the value**
  - **Use the** `parse()` **method to convert a string in the correct pattern to the appropriate object**
  - **Each formatter supports unique format strings and specialized methods**
- **The static** `String.format` **methods are a shortcut to** `java.util.Formatter`
  - **Useful for one-liners**

Inner Classes

Nested Classes

- A class may be nested within another
  - Nested top-level classes
  - Inner classes

Nested Top-level Classes

- Declared within another class but no other block and is declared `static`.
- Has no access to other members of the enclosing class
- As `static` members of the enclosing class top-level classes have scope beyond the enclosing class
  - Not strictly an inner-class
- Useful for defining a class' significant data structures

Inner Classes

- Three varieties of inner classes
  - Member classes
  - Local classes
  - Anonymous classes
- Have visibility of other members of the enclosing class

Member Inner Classes

- Declared within another class but no other block
- Scope is the entire parent in which it is nested
- May be declared private, regular classes are either public or package (default)
- Convenient for implementing event handlers near the listener registration

Local Inner Classes

- Declared within a method of the enclosing class
  - Not accessible outside this method
- Have access to the enclosing methods local variables
  - When the `final` modifier is applied to local variables the variable may be assigned to only once

Anonymous Inner Classes

- Refinement of the local inner class
- Allows a class to be declared with the instance allocation
- Should be limited to class of only a few lines
- Additional instances of the class cannot be created

Anonymous Inner Classes

- **Defined in a block, immediately after a class, or interface constructor specified to the** new **operator**

```
new <SuperType>( <construction params> )
{ <class implementation> }
```

  - **Construction parameters are given to the superclass constructor**

Anonymous Inner Classes
Good, Bad or Just Ugly?

- Convenient, for event handlers and callbacks that need access to parent classes members
- Can be very ugly if the class implementation grows to more than a page of code
- Anything you can do with an anonymous inner class you can do with a local inner class
  - Except absolutely prevent the instantiation of more than one

Another `this` Reference

- Inner classes may access the instance of its class that created it
- The notations is

  <OuterClassName>`.this`
- Member classes should be declared static if this reference is not needed
  - Saves time and space and allows for more efficient garbage collection

Instantiating Inner Classes

- Instances of the inner class may be instantiated outside the outer class
- The notations is

  <OuterClassInstance>.new <InnerClassName>(…)

Instantiating Inner Classes

```
public class Foo {
    private int value;

    public Foo(final int value) {
        this.value = value;
    }

    public class Bar {
        public void process() {
            System.out.println(value);
        }
    }
}
```

```
import Foo.Bar;

public class Driver {
    public static void main(String[] args) {
        Foo f = new Foo(12);
        Bar b = f.new Bar();
        b.process();
    }
}
```

Enumerated Types

- **May have value specific methods**

```
public enum StopLight {
   RED(0xF0000) {
      public String action() {
         return "Stop.";
      }
   },
   AMBER(0xFF9933) {
      public String action() {
          return "Hurry up.";
      }
   },
   GREEN(0x00FF00) {
      public String action() {
         return "Continue on your way.";
      }
   };

   private int rgbColor;

   public abstract String action();
   ...
```

Beware == vs. equals

- The equality operator (==) tests reference equality
  - Identity test, are the two references equal
- The equals method may or may not have this same semantic
  - May provide a value test, consider the String class
    - The equals method tests if the two strings have the same character sequence
    - 99.99999% of the time you want a value test not an identity test

equals

- **Override equals for value equivalency comparison**
  - **The `equals` general contract must be met**
    - **It is *reflexive*:**
      - For any non-null reference value `x`, `x.equals(x)` should return `true`.
    - **It is *symmetric*:**
      - For any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
    - **It is *transitive*:**
      - For any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
    - **It is *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)` should return `false`.**

hashCode

- **The** `hashCode` **and** `equals` **methods must be consistent, i.e. when overriding equals** `hashCode` **generally must be overridden as well**
  - **When overriding** `hashCode`**, its general contract must be met**
    - **Multiple invocations on the same object must consistently return the same integer, provided no information used in** `equals` **comparisons on the object is modified**
    - **If two objects are equal according to the** `equals` **method, then** `hashCode` **must produce the same integer result for both objects.**
    - **If two objects are unequal according to the** `equals` **method, then hashCode need not return distinct integer results for each object.**
      - However, producing distinct integer results for unequal objects may improve the performance of map implementations.

Recipe for hashCode

From "Effective Java"

1. **Store some constant nonzero value, say, 17, in an `int` variable called `result`.**
2. **For each significant field f in your object (each field taken into account by the equals method, that is), do the following:**

   Next slide..

3. **Return result.**

Recipe for hashCode

From "Effective Java"

a) **Compute an `int` hash code `c` for the field:**
   i. **If the field is a `boolean`, compute `(f ? 1 : 0)`.**
   ii. **If the field is a `byte`, `char`, `short`, or `int`, compute `(int) f`.**
   iii. **If the field is a `long`, compute `(int) (f ^ (f >>> 32))`.**
   iv. **If the field is a `float`, compute `Float.floatToIntBits(f)`.**
   v. **If the field is a `double`, compute `Double.doubleToLongBits(f)`, and then hash the resulting long as in step 2.a.iii.**
   vi. **If the field is an object reference and this class's equals method compares the field by recursively invoking equals, recursively invoke `hashCode` on the field. If a more complex comparison is required, compute a "canonical representation" for this field and invoke `hashCode` on the canonical representation. If the value of the field is `null`, return `0` (or some other constant, but `0` is traditional).**
   vii. **If the field is an array, treat it as if each element were a separate field. That is, compute a hash code for each significant element by applying these rules recursively, and combine these values per step 2.b. If every element in an array field is significant, you can use one of the `Arrays.hashCode` methods added in release 1.5.**

b) **Combine the hash code c computed in step 2.a into result as follows:**

```
result = 31 * result + c;
```