Victoria University
f Wellington, New Zealan
Te Whare Wananga o te
Upoko o te Ika a Maui
Aotearoa

SWEN221 Software Development

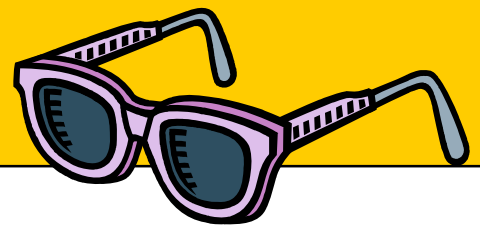
Style

Thomas Kuehne

Victoria University

(slides modified from slides by David J. Pearce & Nicholas Cameron & James Noble & Petra Malik)

Why bother with style?



- Java code may be easy or hard to understand
- The aim of code, comments, diagrams, documentation is **to communicate**
  - With yourself
  - With your team
  - With those who will come after you
- Style guides help to produce code that is easier to read and maintain

Files & Comments

- File Organisation
  - organise them according to Java conventions
  - Eclipse will do this for you
- Comments
  - **`/** Javadoc comment **/, /* */ or //`**
- Tips
  - Good code does not need many comments
    - Good names (methods, variables, classes, fields, etc.) help
    - The trickier the code, the more commenting required
  - Avoid redundant comments
    - E.g. "x = 1; // 1 is assigned to x"
  - Use Javadoc!

Quiz: what's good/bad about this?

```
public class Book { // This class represents a Book
 private String x;
 private String y;

 public Book(String t, String a) {
  x = t;  // set title
  y = a;  // set author
 }
 public String getAuthor() { // Returns Book's Author
  return y;
 }
 public String getTitle() { // Returns the Book's Title
  return x;
}}
```

Layout

- Be consistent
- Use Indentation
- Braces (either beginning or end of lines)
- Order methods logically
  - e.g., **public/private** methods together respectively, access vs manipulation, etc.
- Bad style can make code unreadable

```
int ivl_bi11c1int0n[]; foo.bar
(   x<    ivl_billclint0n,
(0, true))};}} class nextclass { …
```

Two ways to use curly braces

```
int method(int x) {
 int y=3;
 return x+y;
}
```

```
int method(int x)
{
 int y=3;
 return x+y;
}
```

The left version needs less vertical space and thus makes it easier to fit more code on one screen.

The right version may be considered to provide better separation.

Names

- Packages
  - **`lowercase`**
- Classes
  - **`CapsWithWholeWordsCaps (CamelCase)`**
- Exception
  - **`ClassNamesWithException`**
- Interface (when necessary to distinguish from class)
  - **`EndWithI`** or **`Ifc`**
- Class (when necessary to distinguish from interface)
  - **`EndWithImpl`** or **`EndWithObject`**
- Constant (finals)
  - **`UPPERCASE_UNDERSCORE`**
- Avoid hardcoded numbers — use (at least) constants

Names

- Fields
  - **`firstLowerThenCaps`** (or **`trailing_`** or **`thisVar`**)
- Local variables
  - **`firstLowerThenCaps`** (or **`lowercase_with_underscores`**)
- Methods
  - **`firstLowercaseThenCaps`**
- Getters/Setters
  - **`T getX()/setX(T v) OR T x()/x(T v)`**
- Factory/Creator Method
  - **`newT()`**
- Convertor Method
  - **`T toT()`**

Keep variables local

- Always use smallest **scope** possible, i.e. prefer A to B

A

```
class Date {
 int day;

 int nextDay() {
  int r = day + 1;
  return r;
}}
```

B

```
class Date {
 int day;
 int r;

 int nextDay() {
  r = day + 1;
  return r;
}}
```

Others

- Arrays Declaration
  - "`Integer[] x`" (not "`Integer x[]`")
- Guard casts with conditionals
  - E.g. `if(x instanceof C) { C y = (C) x; … } else …`
- Separate *accessors* and *mutators*
  - Otherwise people are forced to mutate
  - E.g. `T pop()` => `void pop() & T top()`
- Avoid "=" inside if- and loop-conditions
  - E.g. `if((x=aMethod()) == 2) { … }`

Example

```
class Date {
 int day;   // day field
 int month; // month field
 int year;  // year field

 int nextDay() {   // next day method
  int r = day + 1; // r is day + 1
  return r;        // return r
}}
```

- What's wrong with this?

Another Example

```
class Date {
 int day, month, year;

 public Date(int day, int month, int year) { … }

 /**
  * Return the day after this one.
  */
 Date nextDay() { return new Date(day+1,month,year); }


 /**
  * Return the day after this one.
  */
 Date prevDay() { return new Date(day-1,month,year); }
}
```

Yet Another Example

```
private Block parseTry(Tree s, FlowGraph g) {
  FlowGraph.Point e = codePoint(null, s);
  Block b = parseStatement(s.getChild(0), null, g);
  Block rb = new Block(b);
  for (int i = 1; i < s.getChildCount(); ++i) {
    Tree c = s.getChild(i);
  if (c.getType() == CATCH) {
    Tree p = c.getChild(0);
    Type.Reference e = (Type.Reference) parseType(param
                                          .getChild(0));
    scs.push(new Scope());
    String n = scs.peek().id + p.getChild(1).getText();
    scs.peek().variables.add(n);
    cfg.add(new FlowGraph.LocalVarDef(n, e, 0, false));
    …
```

- What does this code do ?

Yet Another Example

```
private Block parseTry(Tree stmt, FlowGraph cfg) {
  FlowGraph.Point exit = codePoint(null, stmt);
  Block body = parseStatement(stmt.getChild(0), null, cfg);
  Block rb = new Block(body);
  for (int i = 1; i < stmt.getChildCount(); ++i) {
    Tree child = stmt.getChild(i);
    if (child.getType() == CATCH) {
      Tree param = child.getChild(0);
      Type.Reference exceptionT = (Type.Reference) parseType(param
                                        .getChild(0));
      scopes.push(new Scope());
      String name = scopes.peek().id + param.getChild(1).getText();
      scopes.peek().variables.add(name);
      cfg.add(new FlowGraph.LocalVarDef(name, exceptionT, 0, false));
      …
```

- What does this code do ?

Tools can help ...

- Checkstyle
  - http://checkstyle.sourceforge.net/
- **Jalopy (source code beautifier)**
  - http://jalopy.sourceforge.net/
- **PMD**
  - http://pmd.sourceforge.net/
- **Jlint, FindBugs, etc.**
  - **look for possible bugs in Java code**

SWEN221 Style

- See "**Good Programming Style**" page
  - http://ecs.victoria.ac.nz/Courses/SWEN221_2015T1/StyleGuide
  - Read it and use it
  - You will be marked according to this style!