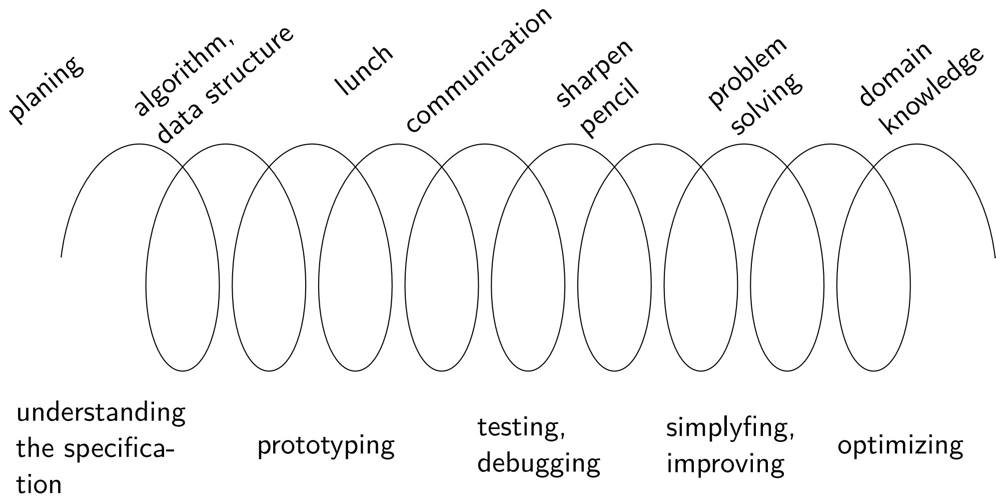
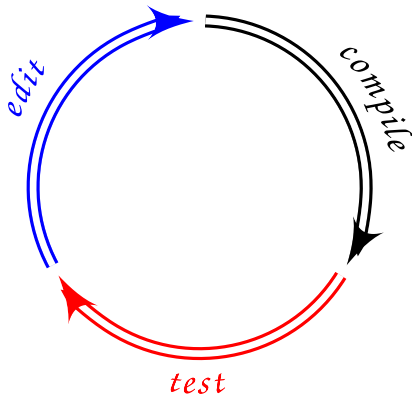
Program Development



Objectives

- editing and refactoring
- errors and warnings
- style
- IDE's
- problem solving

Program Development



IDE (Integrated Development Environment)

IDE's can be complicated to learn, diverse, and single-purpose, yet are valuable, because they:

- support the development process in many ways,
- unify the editing and testing in one application, and
- make development easier, faster, and less error prone.

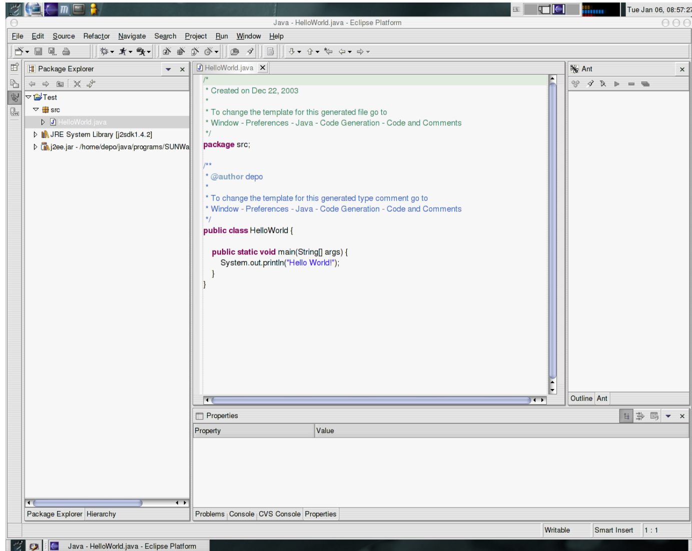
IDE's accomplish these things by hiding the details.

But it is helpful to understand what is going on.

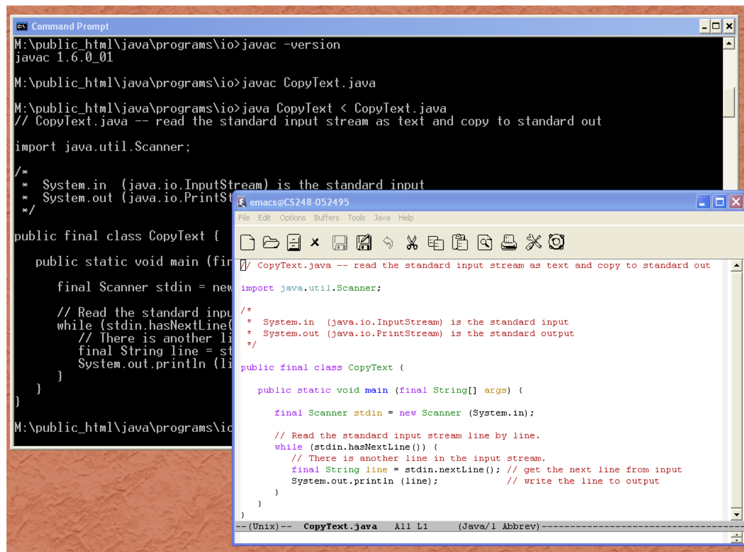
Developing Java Programs – BlueJ

```
/**
 * A class representing students for a simple
 *
 * @author  Michael Kolling
 * @version 1.0, January 1999
 */
class Student extends Person
{
    private String SID;    // student ID numbe

    /**
     * Create a student with default settings
     */
    public Student()
    {
        super("(unknown name)", 0000);
        SID = "(unknown ID)";
    }

    /**
     * Create a student with given name, year
     */
```

Developing Java Programs – Eclipse



Developing Java Programs – Emacs



Developing Java Programs – Notepad++

*new 1 - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

new 1

```
public final class Main {

```

Preferences

General
Editing
New Document
Default Directory
Recent Files History
File Association
Language
Highlighting
Print
Searching
Backup
Auto-Completion
Multi-Instance
Delimiter
Cloud
Search Engine
MISC.

Language Menu

☑ Make language menu compact

Available items	Disabled items
Normal Text	ASP
PHP	Perl
C	NSIS
C++	Ruby
C#	KIXtart
Objective-C	Inno Setup
Java	Fortran (fixed form)
Resource file	BaanC
HTML	ASN.1
XML	AviSynth
Makefile	OScript
Pascal	REBOL
Batch	Visual Prolog
INI file	
MS-DOS Style	

->
<-

☑ Treat backslash as escape character for SQL

Tab Settings

[Default]
normal
actionscript
ada
asm
asn1
asp
autoit
avs
baanc
bash
batch

Tab size : 3
☑ Replace by space

Close

Java source file | length : 31 lines : 2 | Ln : 2 Col : 5 Pos : 32 | Windows (CR LF) | UTF-8 | INS

Developing Java Programs – Intellij

Settings

Editor › Code Style › Java

Scheme: Default IDE

Set from...

- Appearance & Behavior
- Keymap
- Editor
  - General
  - Font
  - Color Scheme
  - Code Style
    - Java
    - ActionScript
    - CFML
    - CoffeeScript
    - CSS
    - Gherkin
    - Groovy
    - GSP
    - Haml
    - HTML
    - JavaScript
    - JSON
    - JSP

Tabs and Indents | Spaces | Wrapping and Braces | Blank Lines | JavaDoc | Imports | Arrangement | Code Generation

- [ ] Use tab character
- [ ] Smart tabs

Tab size: 4

Indent: 4

Continuation indent: 8

- [ ] Keep indents on empty lines

Label indent: 0

- [ ] Absolute label indent
- [ ] Do not indent top level class members
- [ ] Use indents relative to expression start

```
public class Foo {
    public int[] X = new int[]{1, 3, 5, 7, 9, 11};

    public void foo(boolean a, int x, int y, int z) {
        label1:
        do {
            try {
                if (x > 0) {
                    int someVariable = a ? x : y;
                    int anotherVariable = a ? x : y;
                } else if (x < 0) {
                    int someVariable = (y + z);
                    someVariable = x = x + y;
                } else {
                    label2:
                    for (int i = 0; i < 5; i++) doSom
                }
```

- compile error
  - syntax error — `Syntax.java`
  - semantic error — `Semantic.java`
    - type error — `Type.java`
- style error — example program
  Style errors are mistakes in the program source code that contravene policy or hamper the ability of programmers to read and understand the program even though the program can be translated by the compiler into a executable program. A list of errors
- execution error or (fatal) runtime error — example program
  Runtime errors are mistakes that manifest themselves during the execution of the program. These errors prevent the computer from completing the execution of the program.
- logic error — example program
  Logic errors are mistakes in the behavior of the program even though the program can be translated into a running, executable program.

Java requires many suspicious behaviors to be flagged as errors (not just warnings). According to the Java Language Specification:
"It is a compile-time error if a statement cannot be executed because it is unreachable."

In many languages suspicious code is given a warning, but the program may be executed anyway.

Warnings, as opposed to compile-time errors, have gradually been added to the Java language specification.

Java has optional warnings enabled by `javac -Xlint`
In Java 1.6 the complete list was

```
cast,deprecation,divzero,empty,unchecked,
fallthrough,path,serial,finally,overrides
```

The warnings `deprecation` and `unchecked` are checked in all cases (regardless of the command line options).

```
java -Xlint:all -Xlint:-serial -Werror
```

Thou ſhalt lint thy program

Thou ſhalt lint thy program

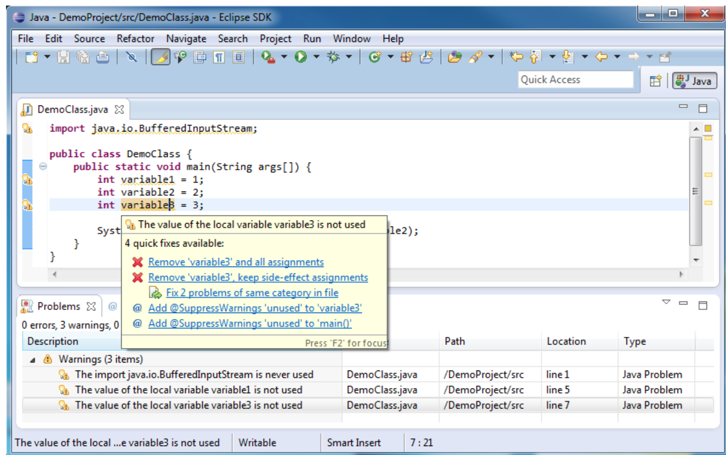
It is common for software development groups to require `-Xlint` (enable warnings) and `-Werror` (treat warngins as errors) for `javac` in order to insure the code is warning-free.

javac warnings

```
$javac -X   [Java 16]
cast                use of unnecessary casts.
classfile           issues related to classfile contents.
deprecation         use of deprecated items.
dep-ann             missing @Deprecated annotation.
divzero             division by constant integer 0.
empty               empty statement after if.
fallthrough         falling through from a case of a switch statement.
finally             finally clauses that do not terminate normally.
options             issues relating to use of command line options.
overrides           issues regarding method overrides.
path                invalid path elements on the command line.
rawtypes            use of raw types.
serial              Serializable classes with no serial version ID.
static              accessing a static member using an instance.
try                 issues relating to use of try blocks.
unchecked           unchecked operations.
varargs             potentially unsafe vararg methods
```

javac warnings

```
$ javac --help-lint
The supported keys for -Xlint are:
    auxiliaryclass       Warn about an auxiliary class that is hidden in a source file, and is used from other files.
    cast                 Warn about use of unnecessary casts.
    classfile            Warn about issues related to classfile contents.
    deprecation          Warn about use of deprecated items.
    dep-ann              Warn about items marked as deprecated in JavaDoc but not using the @Deprecated annotation.
    divzero              Warn about division by constant integer 0.
    empty                Warn about empty statement after if.
    exports              Warn about issues regarding module exports.
    fallthrough          Warn about falling through from one case of a switch statement to the next.
    finally              Warn about finally clauses that do not terminate normally.
    missing-explicit-ctor Warn about missing explicit constructors in public and protected classes in exported packages.
    module               Warn about module system related issues.
    opens                Warn about issues regarding module opens.
    options              Warn about issues relating to use of command line options.
    overloads            Warn about issues regarding method overloads.
    overrides            Warn about issues regarding method overrides.
    path                 Warn about invalid path elements on the command line.
    processing           Warn about issues regarding annotation processing.
    rawtypes             Warn about use of raw types.
    removal              Warn about use of API that has been marked for removal.
    requires-automatic   Warn about use of automatic modules in the requires clauses.
    requires-transitive-automatic Warn about automatic modules in requires transitive.
    static               Warn about accessing a static member using an instance.
    strictfp             Warn about unnecessary use of the strictfp modifier.
    synchronization      Warn about synchronization attempts on instances of value-based classes.
    text-blocks          Warn about inconsistent white space characters in text block indentation.
    try                  Warn about issues relating to use of try blocks (i.e. try-with-resources).
    unchecked            Warn about unchecked operations.
    varargs              Warn about potentially unsafe vararg methods.
    preview              Warn about use of preview language features.
```



Eclipse warns about semantic problems not required by the Java language specification

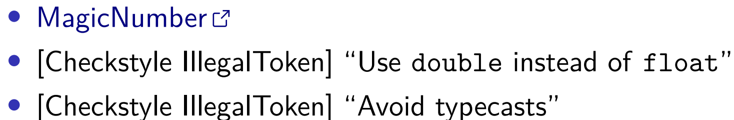
If you make a mistake and write a program that goes into an endless loop, and the computer runs out time or space resources and terminates your program prematurely, is this a runtime or a logic error?

Either, both, what difference does it make?

What is a compiler warning (as opposed to an error)?

Have you ever encountered a compiler warning issued by `javac`?

Indenting is very important; many annoying white-space complaints



- MagicNumber
- [Checkstyle IllegalToken] “Use double instead of float”
- [Checkstyle IllegalToken] “Avoid typecasts”

```java
Integer.parseInt ("42");   // String to int
Integer.valueOf ("42");    // String to Integer
Double.parseDouble("42"); // String to double
Double.valueOf (42);       // int or double to
                      // Double [double, auto-boxing]
Math.round(3.4D) // double to long
Math.ceil(3.4D)  // double to double!
Math.floor(3.4D) // double to double!
Math.floorDiv (42L,43L) // long,long -> long
```

```java
/* Coerce to double, create Double object,
   auto-unbox, discard object; lots of overhead
 */
double d = Double.valueOf (42);
```

```java
/* Deprecated because new immutable records are more
   efficient than plain, old Java classes.
 */
Double d = new Double (42);
```

Java API doc Math

No good explicit function to convert a primitive integer to a primitive double, e.g., `Real(42)` in Ada, `fromIntegral(42)` in Haskell.

```
double x = 5L; // sometimes works
double x = 5;
float y = 5L;
float y = 5;
```

A cast (implicit widening conversion) could be

```
double quotient = (double) 42 / 5;  // Avoid cast

double meaningOfLife = 42; // some int or long expression
double quotient = meaningOfLife / 5.0D;

long x = Math.round (5.3D);
```

```
jshell> double x = 5L;
x ==> 5.0

jshell> double x = 5555555555555555555L;
x ==> 5.5555555555555553E18

jshell> long x =  round (ceil (45.3D))
x ==> 46

jshell> long x =  round (ceil (45.3F)))
x ==> 46

jshell> int x = toIntExact (round (ceil (45.3D)))
x ==> 46
```

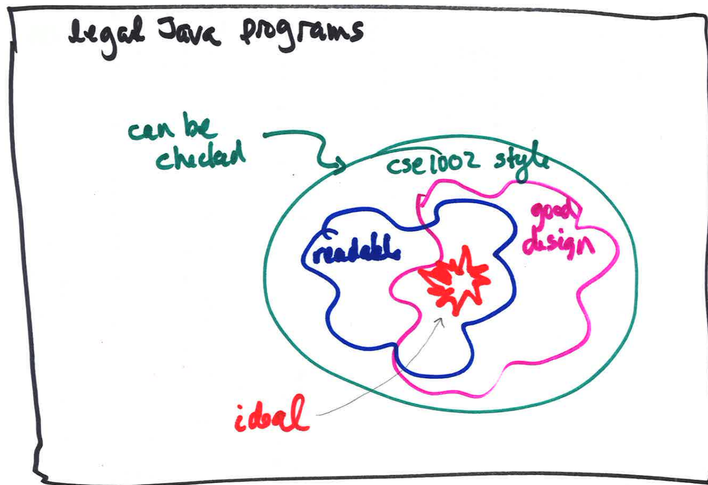
Thou ſhalt not uſe a caſt

A case is a type name in parentheses,
e.g., `(int) 4.5D`
Avoid mistakes by carefully converting
from one data type to another

Thou ſhalt indent by three

(Four is perfectly reasonably, but we cannot check for three *or* four.)

Ideal Programs



Ideal programs are readable and well-designed

Editing versus refactoring.

Definition

Refactoring code is the process of restructuring existing code with knowledge of the programming language (e.g., the scope of identifiers), usually keeping the same behavior.

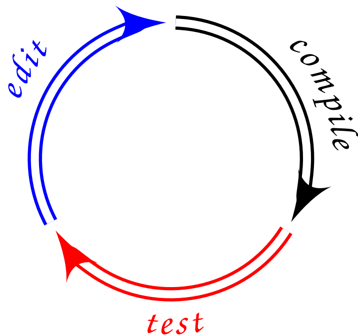
The intention is usually to improve the design, efficiency, or readability of the code. Refactoring code is “smart” editing.

“Dumb” editing text is oblivious to the structure, semantics, and behavior of the text, like replacing all occurences of the letter ’a’ in a source program with the letter ’b’. This will likely create many syntax errors.

“Smart” editing (refactoring) code respects the structure, semantics, and behavior of the code, like replacing all uses of the identifier ’a’ in a source program with the identifier ’b’.

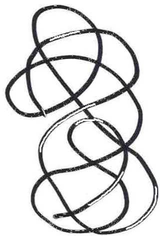
Many IDEs can perform intelligent changes like renaming identifiers, introducting methods, adding parameters to methods, adding declarations to remove magic numbers, and so on.

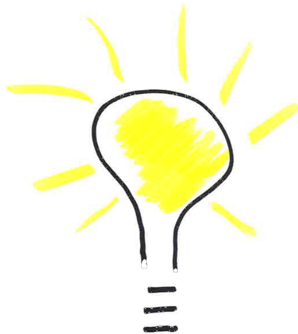
Program Development



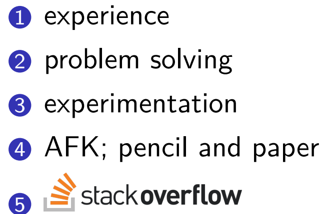
At what point does planning and thinking come in?

... understanding the requirements?





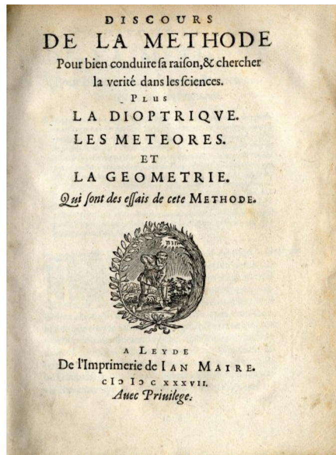
Where do ideas come from?

1. experience
2. problem solving
3. experimentation
4. AFK; pencil and paper
5. 

S&W Lessons, Page 318ff

- Expect bugs
- Keep modules small
- Limit interactions
- Develop code incrementally
- Solve an easier problem
- Consider a recursive solution
- Build tools where appropriate
- Reuse software when possible

Problem Solving



DISCOURS
DE LA METHODE
Pour bien conduire ſa raiſon,& chercher
la verité dans les ſciences.
Plus
LA DIOPTRIQVE.
LES METEORES.
ET
LA GEOMETRIE.
Qui ſont des eſſais de cete METHODE.

A LEYDE
De l'Imprimerie de IAN MAIRE.
cIↄ Iↄ c xxxvii.
Auec Priuilege.

René Descartes (1596–1650)
Discours de la méthode, 1637

1. Never assume, be critical, put aside your preconceived notions

   Le premier était de ne recevoir jamis aucune chose pour vraie que je ne la connusse évidemment être telle;

2. Decompose your problem until each piece becomes trivial.
3. Solve the simplest things first.
4. Keep revising your work so that nothing is forgotten.

Computational Thinking

1. Define. Manageable questions
2. Abstract. Transform into precise form
3. Compute. Identify and resolve issues
4. Interpret. Re-contextualize and refine