Program Development

- Planning
- Algorithm, data structure
- Communication
- Sharpen pencil
- Problem solving
- Domain knowledge

- Understanding the specification
- Prototyping
- Testing, debugging
- Simplyfing, improving
- Optimizing
Objectives

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

```java
/**
 * A class representing students for a simple database.
 * @author Michael Kolling
 * @version 1.0, January 1999
 */
class Student extends Person {
    private String SID; // student ID number
    /**
     * Create a student with default settings.
     * @return
     */
    public Student() {
        super("(unknown name)", 0000);
        SID = ":(unknown ID)"
    }
    /**
     * Create a student with given name, year.
     * @param name
     * @param year
     */
    public Student(String name, int year) {
        super(name, year);
        SID = ":(unknown ID)"
    }
}
```

Diagram showing class Student inheriting from Person and relationships with other classes such as Staff and Database.
Developing Java Programs – Eclipse
Developing Java Programs – Emacs

```
import java.util.Scanner;

public final class CopyText {
    public static void main(final String[] arg) {
        final Scanner stdin = new Scanner(System.in);
        while (stdin.hasNextLine()) {
            System.out.println(stdin.nextLine());
        }
    }
}
```
Developing Java Programs – Notepad++

![Image of Notepad++ interface with Java code and settings]

- `public final class Main {`  
- Language Menu:
  - Make language menu compact
  - Disabled items:
    - Normal Text
    - PHP
    - C++
    - C#
    - Objective-C

- Tab Settings:
  - Default
  - Replace by space

- Java source file length: 31 lines: 2
  - Ln: 2 Col: 5 Pos: 32
  - Windows (CR LF) UTF-8 INS
Developing Java Programs – Intellij
• 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 warnings as errors) for `javac` in order to insure the code is warning-free.
<table>
<thead>
<tr>
<th>javac warnings</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$javac -X</td>
<td>[Java 16]</td>
</tr>
<tr>
<td>cast</td>
<td>use of unnecessary casts.</td>
</tr>
<tr>
<td>classfile</td>
<td>issues related to classfile contents.</td>
</tr>
<tr>
<td>deprecation</td>
<td>use of deprecated items.</td>
</tr>
<tr>
<td>dep-ann</td>
<td>missing @Deprecated annotation.</td>
</tr>
<tr>
<td>divzero</td>
<td>division by constant integer 0.</td>
</tr>
<tr>
<td>empty</td>
<td>empty statement after if.</td>
</tr>
<tr>
<td>fallthrough</td>
<td>falling through from a case of a switch statement.</td>
</tr>
<tr>
<td>finally</td>
<td>finally clauses that do not terminate normally.</td>
</tr>
<tr>
<td>options</td>
<td>issues relating to use of command line options.</td>
</tr>
<tr>
<td>overrides</td>
<td>issues regarding method overrides.</td>
</tr>
<tr>
<td>path</td>
<td>invalid path elements on the command line.</td>
</tr>
<tr>
<td>rawtypes</td>
<td>use of raw types.</td>
</tr>
<tr>
<td>serial</td>
<td>Serializable classes with no serial version ID.</td>
</tr>
<tr>
<td>static</td>
<td>accessing a static member using an instance.</td>
</tr>
<tr>
<td>try</td>
<td>issues relating to use of try blocks.</td>
</tr>
<tr>
<td>unchecked</td>
<td>unchecked operations.</td>
</tr>
<tr>
<td>varargs</td>
<td>potentially unsafe vararg methods</td>
</tr>
</tbody>
</table>
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”
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
/* Coerce to double, create Double object, auto-unbox, discard object; lots of overhead */
double d = Double.valueOf(42);

/* 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., \texttt{Real(42)} in Ada, \texttt{fromIntegral(42)} in Haskell.

\begin{verbatim}
double x = 5L; // sometimes works
double x = 5;
float y = 5L;
float y = 5;
\end{verbatim}

A cast (implicit widening conversion) could be

\begin{verbatim}
double quotient = (double) 42 / 5; // Avoid cast
\end{verbatim}

\begin{verbatim}
double meaningOfLife = 42; // some int or long expression
double quotient = meaningOfLife / 5.0D;
\end{verbatim}

\begin{verbatim}
long x = Math.round (5.3D);
\end{verbatim}
jshell> `double` x = 5L;
x ==> 5.0

jshell> `double` x = 5555555555555555555 L;
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 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 occurrences 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, introducing methods, adding parameters to methods, adding declarations to remove magic numbers, and so on.
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. stack overflow
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

1. Never assume, be critical, put aside your preconceived notions
   
   *Le premier était de ne recevoir jamais 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.

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

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

Why learn computational thinking?
- Because it helps you solve problems—and is increasingly critical for individuals and organizations as computation becomes more effec...