Developing Java Programs – BlueJ

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

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

    /**
     * Create a student with given name, year, address.
     */
    public Student(String name, int year, Address addr) {
        super(name, year, addr);
    }

    // getters and setters
}
```
Developing Java Programs – Eclipse
Developing Java Programs – Intellij
• compile error
  • syntax error — Syntax.java
  • semantic error — Semantic.java
  • type error — Type.java

• Eclipse warnings

• 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.”

Java has optional warnings enabled by `javac -Xlint`

In Java 1.6 the complete list (obtained by `javac -X`):

`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`
javac warnings

$javac -X [Java 16]
cast         use of unnecessary casts.
classfile    issues related to classfile contents.
deprecation  use of deprecated items.
dep-ann

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
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.

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

A cast (implicit widening conversion) could be

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

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

```java
 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 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), this 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 declares 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. stackoverflow
• 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 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.

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