Really Bad Java code

```
System.out.print("Flight ");
System.out.print("819");
System.out.print(" to ");
System.out.print("Tokyo");
System.out.print(" takes off at ");
System.out.print("8:50 AM");
System.out.println();
System.out.println();
System.out.println();

System.out.print("Flight ");
System.out.print("211");
System.out.print(" to ");
System.out.print("New York");
System.out.print(" takes off at ");
System.out.print("8:55 AM");
System.out.println();
System.out.println();
System.out.println();
// and 10 more lines each for 350 flights
```

Code Duplication is Evil

Abstraction
is
Good

Incremental improvement

```
static void skipThreeLines () {
    System.out.println();
    System.out.println();
    System.out.println();
}
```

Fairly Bad Java code

```
System.out.print("Flight ");
System.out.print("819");
System.out.print(" to ");
System.out.print("Tokyo");
System.out.print(" takes off at ");
System.out.print("8:50 AM");
skipThreeLines ();

System.out.print("Flight ");
System.out.print("211");
System.out.print(" to ");
System.out.print("New York");
System.out.print(" takes off at ");
System.out.print("8:55 AM");
skipThreeLines ();
// and 7 more lines each for 350 flights
```

Conceptual clarity and Kolmogorov Complexity

Conceptual clarity:

- Well-named functions clarify the programmer's intent
- Easier to maintain

Conceptual clarity and Kolmogorov Complexity

Conceptual clarity:

- Well-named functions clarify the programmer's intent
- Easier to maintain

Kolmogorov Complexity:

- The (Kolmogorov) complexity of a program is the length of the shortest program which does the same thing.
- Represents a stable notion of 'information content'
- Before: $10 \times 350 = 3500$ lines. After: $5 + 7 \times 350 = 2455$ lines.

Conceptual clarity and Kolmogorov Complexity

Conceptual clarity:

- Well-named functions clarify the programmer's intent
- Easier to maintain

Kolmogorov Complexity:

- The (Kolmogorov) complexity of a program is the length of the shortest program which does the same thing.
- Represents a stable notion of 'information content'
- Before: $10 \times 350 = 3500$ lines. After: $5 + 7 \times 350 = 2455$ lines.

Remember: programs which are too information-sparse are evil, programs which are too dense are extremely hard to maintain.

Getting better

```
static void printTakeOffTime( String flightNum,
  String destination, String time) {
    System.out.print("Flight ");
    System.out.print(flightNum);
    System.out.print(" to ");
    System.out.print(destination);
    System.out.print(" takes off at ");
    System.out.print(time);
    skipThreeLines ();
}
```

Getting better

```
static void printTakeOffTime( String flightNum,
  String destination, String time) {
    System.out.print("Flight ");
    System.out.print(flightNum);
    System.out.print(" to ");
    System.out.print(destination);
    System.out.print(" takes off at ");
    System.out.print(time);
    skipThreeLines ();
}

printTakeOffTime("819", "Tokyo", "8:50 AM");
printTakeOffTime("221", "New York", "8:55 AM");
// and 1 more line each for 350 flights
```

Getting better

```
static void printTakeOffTime( String flightNum,
  String destination, String time) {
    System.out.print("Flight ");
    System.out.print(flightNum);
    System.out.print(" to ");
    System.out.print(destination);
    System.out.print(" takes off at ");
    System.out.print(time);
    skipThreeLines ();
}

printTakeOffTime("819", "Tokyo", "8:50 AM");
printTakeOffTime("221", "New York", "8:55 AM");
// and 1 more line each for 350 flights
```

$5 + 9 + 350 \times 1 = 364$ lines.

Now we're talking!

```
static void printTakeOffTime( String flightNum,
  String destination, String time) {
    System.out.printf("Flight␣%s␣to␣%s␣takes␣off␣at␣%s",
        flightNum, destination, time);
    skipThreeLines ();
}
```

Now we're talking!

```
static void printTakeOffTime( String flightNum,
  String destination, String time) {
    System.out.printf("Flight %s to %s takes off at %s",
        flightNum, destination, time);
    skipThreeLines ();
}

printTakeOffTime("819", "Tokyo", "8:50 AM");
printTakeOffTime("221", "New York", "8:55 AM");
// and 1 more line each for 350 flights
```

Now we're talking!

```
static void printTakeOffTime( String flightNum,
  String destination, String time) {
    System.out.printf("Flight %s to %s takes off at %s",
        flightNum, destination, time);
    skipThreeLines ();
}

printTakeOffTime("819", "Tokyo", "8:50 AM");
printTakeOffTime("221", "New York", "8:55 AM");
// and 1 more line each for 350 flights
```

$5 + 5 + 350 \times 1 = 360$ lines.

Now we're talking!

```
static void printTakeOffTime( String flightNum,
  String destination, String time) {
    System.out.printf("Flight %s to %s takes off at %s",
        flightNum, destination, time);
    skipThreeLines ();
}

printTakeOffTime("819", "Tokyo", "8:50 AM");
printTakeOffTime("221", "New York", "8:55 AM");
// and 1 more line each for 350 flights
```

$5 + 5 + 350 \times 1 = 360$ lines.

What is still wrong with our code?

Ocaml

```
let printTakeOffTime flightNum destination time =
  Printf.fprintf stdout "Flight␣%s␣to␣%s␣takes␣off␣at␣%s\n\n\n"
        flightNum destination time ;;

printTakeOffTime "819" "Tokyo" "8:50␣AM␣";;
```

Return values

```
a = 3;
b = 4;
c = 5;
d = 12;
u = Math.sqrt(a * a + b * b);
v = Math.sqrt(c * c + d * d);
```

Return values

```
a = 3;
b = 4;
c = 5;
d = 12;
u = Math.sqrt(a * a + b * b);
v = Math.sqrt(c * c + d * d);

static double hypothenuse (final double x, final double y) {
    return Math.sqrt(x * x + y * y); }
```

Return values

```
a = 3;
b = 4;
c = 5;
d = 12;
u = Math.sqrt(a * a + b * b);
v = Math.sqrt(c * c + d * d);

static double hypothenuse (final double x, final double y) {
    return Math.sqrt(x * x + y * y); }

u = hypothenuse(a, b);
v = hypothenuse(c, d);
```

Return values

```
a = 3;
b = 4;
c = 5;
d = 12;
u = Math.sqrt(a * a + b * b);
v = Math.sqrt(c * c + d * d);

static double hypothenuse (final double x, final double y) {
    return Math.sqrt(x * x + y * y); }

u = hypothenuse(a, b);
v = hypothenuse(c, d);

let hypothenuse x y = sqrt (x *. x +. y *. y)
```

The return construct

```
// doesn't work:
static double hypothenuse (final double x, final double y) {
    Math.sqrt(x * x + y * y); }
// works:
static double hypothenuse (final double x, final double y) {
    return Math.sqrt(x * x + y * y); }
```

The return construct

```
static int sign (final int x) {
  if (x < 0) return -1;
  else if (x == 0) return 0;
  else return 1; }
```

The return construct

```
static int sign (final int x) {
  if (x < 0) return -1;
  else if (x == 0) return 0;
  else return 1; }
```

Can be written as

```
static int sign (final int x) {
  if (x < 0) return -1;
  if (x == 0) return 0;
  return 1; }
```

The return construct

```
static int sign (final int x) {
  if (x < 0) return -1;
  else if (x == 0) return 0;
  else return 1; }
```

Can be written as

```
static int sign (final int x) {
  if (x < 0) return -1;
  if (x == 0) return 0;
  return 1; }
```

Think about:

```
  int x = 3;
  return 5;
  x = 5;
```

functions and procedures

- A *function*: return a value
- A *procedure*: does not return a value
- A function call: used as an *expression*
- A procedure call: used an a *statement*
- Considering functions calls as statements are in bad form

Global Variables

Suppose you have first defined

```
static void reset() { x = 0; }
```

then later

```
int x = 3;
x = 0;
reset ();
x = 5;
```

Global Variables

Suppose you have first defined

```
static void reset() { x = 0; }
```

then later

```
int x = 3;
x = 0;
reset ();
x = 5;
```

This way works:

```
static int x = 3;
static void reset() { x = 0; }
```

and then somewhere else:

```
x = 5;
reset ();
```

Global Variables

Suppose you have first defined

```
static void reset() { x = 0; }
```

then later

```
int x = 3;
x = 0;
reset ();
x = 5;
```

This way works:

```
static int x = 3;
static void reset() { x = 0; }
```

and then somewhere else:

```
x = 5;
reset ();
```

Convenience: $\uparrow\uparrow$ local reasoning $\downarrow\downarrow\downarrow\downarrow$

Main Program

- Global variable declarations
- Function declarations and definitions
- main program (statement)

Main Program

- Global variable declarations
- Function declarations and definitions
- main program (statement)

In Java, wrapped in a class.

```
class Foo {
  static int x = 3;
  static void reset() { x = 0; }

  public static void main(String[] args) {
    x = 5;
    reset () ;
  }
```

Main Program

In C:

```
int x = 5;
void reset() { x = 0; }
int main() {
    x = 5;
    reset();
    return 0;
}
```

Main Program

In C:

```
int x = 5;
void reset() { x = 0; }
int main() {
    x = 5;
    reset();
    return 0;
}
```

In Ocaml:

```
let x = ref 5
let reset _ = x := 0

let _ = x := 5;
        reset ()
```

Shadowing in Java (and C)

```
class Prog {
  static int n;

  static int f (final int x) {
    int p = 5;
    return n + p + x;
  }

  static int g (final int x) {
    int n = 5;
    return n + n + x;
  }

  public static void main (String[] args) {
    n = 4;
    System.out.println(f(6));
    System.out.println(g(6));
  }
}
```

Shadowing in ML

```
let x = 5 in
  let x = 6 in
  x + x
```

Shadowing in ML

```
let x = 5 in
  let x = 6 in
  x + x
```

Note: the above is shadowing, whereas in Java

```
int x;
x = 5;
x = 6;
return x+x;
```

works by assignment.

Overloading

```
static int f (final int x) { return x; }
static int f (final int x) { return x+1; }
```

is illegal.

Overloading

```
static int f (final int x) { return x; }
static int f (final int x) { return x+1; }
```

is illegal.

```
static int f (final int x) { return x; }
static int f (final int x, final int y) { return x+y; }
static int f (final boolean x) { return 7; }
```

is fine.

Overloading

```
static int f (final int x) { return x; }
static int f (final int x) { return x+1; }
```

is illegal.

```
static int f (final int x) { return x; }
static int f (final int x, final int y) { return x+y; }
static int f (final boolean x) { return 7; }
```

is fine.

A **signature** is a combination of a **name** and a **sequence of argument types**. In Java, each unique signature can be defined, thus *overloading* a name.

Overloading

```
static int f (final int x) { return x; }
static int f (final int x) { return x+1; }
```

is illegal.

```
static int f (final int x) { return x; }
static int f (final int x, final int y) { return x+y; }
static int f (final boolean x) { return 7; }
```

is fine.

A **signature** is a combination of a **name** and a **sequence of argument types**. In Java, each unique signature can be defined, thus *overloading* a name.

No overloading in C or Ocaml.

Overloading

```
static int f (final int x) { return x; }
static int f (final int x) { return x+1; }
```

is illegal.

```
static int f (final int x) { return x; }
static int f (final int x, final int y) { return x+y; }
static int f (final boolean x) { return 7; }
```

is fine.

A **signature** is a combination of a **name** and a **sequence of argument types**. In Java, each unique signature can be defined, thus *overloading* a name.

No overloading in C or Ocaml.

Commentary: overloading is extremely convenient. Except in the presence of sub-typing, when it becomes very complicated. Java has sub-typing.