

What does this program print?

Attempt 1

```
public class Rec {  
  private static int f(int  
  x) {  
    if(x<2) {      return  
      1;  
    } return f(x-  
    1)+f(x-2  
    );}  
  public static void main(  
    String[] args) {  
    System.out.println(f(5));}}
```

What does this program print?

Attempt 2: A hint...

```
public class Rec {  
    private static int f(int x) {  
        if (x < 2) {  
            return 1;  
        }  
        return f(x - 1) + f(x - 2);  
    }  
  
    public static void main(String[] args) {  
        System.out.println(f(5));  
    }  
}
```

What does this class do?

Attempt 1

```
public static final class Oc {  
    private final Object[] e  
        = new Object[1000000];  
    private int pe = -1;  
    private int po = 0;  
  
    public void a(Object x) {  
        e[po++] = x;  
    }  
  
    public Object b() {  
        return e[pe++];  
    }  
}
```

What does this class do?

Attempt 2: A hint...

```
public static final class Queue {  
    private final Object[] myValues  
        = new Object[BIG_VALUE];  
    private int myHead = -1;  
    private int myTail = 0;  
  
    public void enqueue(Object x) {  
        myValues[myTail++] = x;  
    }  
  
    public Object dequeue() {  
        return myValues[myHead++];  
    }  
}
```

A bug fixed

class Queue

```
private int myHead = -1;  
private int myTail = 0;  
  
public void enqueue(Object x) {  
    myValues[myTail] = x;  
    myTail++;  
}  
  
public Object dequeue() {  
    myHead++;  
    return myValues[myHead];  
}
```

What does this function do?

A humble two-line function 😊

```
boolean p(int x) {  
    int y = x * (030 >> 4 << 030);  
    return y == 0;  
}
```

What does this function do?

A humble two-line function 😊

```
boolean p(int x) {  
    int y = x * (030 >> 4 << 030);  
    return y == 0;  
}
```

Hints

- ▶ $a \ll s = a * 2^s$ — bitwise shift left
- ▶ $a \gg s = a / 2^s$ — bitwise shift right
- ▶ `0x<DIGITS>` — hexadecimal number
- ▶ `0<DIGITS>` — octal number

What we have (hopefully) learned so far

What we have (hopefully) learned so far

Example 1: Fibonacci numbers

Format your programs properly!

What we have (hopefully) learned so far

Example 1: Fibonacci numbers

Format your programs properly!

Example 2: Queue

Give understandable names to program elements!

What we have (hopefully) learned so far

Example 1: Fibonacci numbers

Format your programs properly!

Example 2: Queue

Give understandable names to program elements!

- ▶ A good program does not demand comments other than JavaDoc for interfaces.

What we have (hopefully) learned so far

Example 1: Fibonacci numbers

Format your programs properly!

Example 2: Queue

Give understandable names to program elements!

- ▶ A good program does not demand comments other than JavaDoc for interfaces.

Example 3: Divisibility by 256

Do not outsmart yourself!

What we have (hopefully) learned so far

Example 1: Fibonacci numbers

Format your programs properly!

Example 2: Queue

Give understandable names to program elements!

- ▶ A good program does not demand comments other than JavaDoc for interfaces.

Example 3: Divisibility by 256

Do not outsmart yourself!

Use understandable code constructs.

Coding Conventions for Java

Andrey Breslav

ITMO University, St. Petersburg / University of Tartu

Oct. 30, 2009

Exercise 1

- ▶ Groups: 4 people each

Exercise 1

- ▶ Groups: 4 people each
- ▶ Time: 10 minutes

Exercise 1

- ▶ Groups: 4 people each
- ▶ Time: 10 minutes
- ▶ Task: come up with 2 rules (conventions)

Exercise 1

- ▶ Groups: 4 people each
- ▶ Time: 10 minutes
- ▶ Task: come up with 2 rules (conventions)
 - ▶ Clear formulations

Exercise 1

- ▶ Groups: 4 people each
- ▶ Time: 10 minutes
- ▶ Task: come up with 2 rules (conventions)
 - ▶ Clear formulations
 - ▶ Clear explanations

“Code Conventions for Java™ Programming Language”

<http://java.sun.com/docs/codeconv/>

- ▶ A “standard” provided by the Sun Microsystems
- ▶ Reflects the opinion of the creators of the language
- ▶ Adopted by many projects

Naming Conventions: Types and Methods

Classes and Interfaces

- ▶ Class name is most likely a noun phrase
- ▶ Written in CamelCase
 - ▶ MyFavouriteClass
- ▶ [Not in CCJ]: Interface names are prefixed with “I”: IModel

Methods

- ▶ Method name is most likely a verb phrase
- ▶ Starts with a lower case letter, then — CamelCase
 - ▶ doTheJob()
- ▶ Common prefixes “get”, “is”, “set”

Naming Conventions: Variables and Constants

Variables: fields, local variables, parameters

- ▶ Starts with a lower case letter, then — CamelCase
 - ▶ counter, firstOccurrence
- ▶ Names should never start with “\$” or “_”
- ▶ [Not in CCJ]: field names are prefixed with “my”

Constants: **static final**, normally **public**

- ▶ Uppercase letters, words separated by underscores “_”
 - ▶ THE_CONSTANT

Naming Conventions: Packages

Packages

- ▶ Only lowercase ASCII letters
- ▶ Starts with a unique domain name (reversed)
 - ▶ `org.eclipse.emf.ecore`

Declaration Order: Elements of a Class

Only one top-level class should be declared in a file

Order of elements in a class

1. Nested classes
2. Static fields
3. Static methods
4. Constructors
5. Instance fields
6. Instance methods

Declaration Order: Inside a Method

Only one variable per line (applies also to fields)

Bad

```
int a = 0, b = 3;
```

Good

```
int a = 0;  
int b = 3;
```

Variables are initialized upon declaration (if possible)

Bad

```
int a;
```

Good

```
int a = 0;
```

[Not in CCJ]: Variables are defined as close to their usage as possible

Declaration Order: Inside a Method

Only one variable per line (applies also to fields)

Bad

```
int a = 0, b = 3;
```

Good

```
int a = 0;  
int b = 3;
```

Variables are initialized upon declaration (if possible)

Bad

```
int a;
```

Good

```
int a = 0;
```

[Not in CCJ]: Variables are defined as close to their usage as possible

Actually CCJ recommends the opposite

Declaration Order: Variable Declaration Example

Bad

```
int a = 0;
int b = 5;
while (b > 0) {
    a = b * b;
    if (a > 10) {
        c++;
    }
}
```

Good

```
int b = 5;
while (b > 0) {
    int a = b * b;
    if (a > 10) {
        c++;
    }
}
```

Declaration Order: Warning

May cause a problem

```
int b = 5;
while (b > 0) {
    A a = new A();
    if (a.getV(b) > 10) {
        c++;
    }
}
```

Safe

```
int b = 5;
while (b > 0) {
    int a = b * b;
    if (a > 10) {
        c++;
    }
}
```

Declaration Order: Declaring Arrays

Brackets are put after the **element type**

Bad

```
int a[] = 1, 2, 3;
```

Good

```
int[] a = 1, 2, 3;
```

Whitespace: Where to Put Spaces

- ▶ Before an opening curly brace (“{”):
public void method() {

Whitespace: Where to Put Spaces

- ▶ Before an opening curly brace (“{”):
`public void method() {`
- ▶ After a comma (“,”):
`public void method(int a, int b)`

Whitespace: Where to Put Spaces

- ▶ Before an opening curly brace (“{”):
`public void method() {`
- ▶ After a comma (“,”):
`public void method(int a, int b)`
- ▶ After a control operator keyword (e.g., `if`):
`if (cond) {`

Whitespace: Where to Put Spaces

- ▶ Before an opening curly brace (“{”):
`public void method() {`
- ▶ After a comma (“,”):
`public void method(int a, int b)`
- ▶ After a control operator keyword (e.g., **if**):
`if (cond) {`
- ▶ After a semicolon (“;”) inside **for** loop header:
`for (int i = 0; i < 10; i++) {`

Whitespace: Where to Put Spaces

- ▶ Before an opening curly brace (“{”):
public void method() {
- ▶ After a comma (“,”):
public void method(**int** a, **int** b)
- ▶ After a control operator keyword (e.g., **if**):
if (cond) {
- ▶ After a semicolon (“;”) inside **for** loop header:
for (**int** i = 0; i < 10; i++) {
- ▶ Around binary operations (e.g., “+” and “/”):
int a = a + b * (c - 1 / 2.0);

Whitespace: Where to Put Spaces

- ▶ Before an opening curly brace (“{”):
public void method() {
- ▶ After a comma (“,”):
public void method(**int** a, **int** b)
- ▶ After a control operator keyword (e.g., **if**):
if (cond) {
- ▶ After a semicolon (“;”) inside **for** loop header:
for (**int** i = 0; i < 10; i++) {
- ▶ Around binary operations (e.g., “+” and “/”):
int a = a + b * (c - 1 / 2.0);
- ▶ In a ternary operator (... ? ... : ...):
(a > b) ? a : b;

Whitespace: Where to Put Spaces

- ▶ Before an opening curly brace (“{”):
`public void method() {`
- ▶ After a comma (“,”):
`public void method(int a, int b)`
- ▶ After a control operator keyword (e.g., **if**):
`if (cond) {`
- ▶ After a semicolon (“;”) inside **for** loop header:
`for (int i = 0; i < 10; i++) {`
- ▶ Around binary operations (e.g., “+” and “/”):
`int a = a + b * (c - 1 / 2.0);`
- ▶ In a ternary operator (... ? ... : ...):
`(a > b) ? a : b;`
- ▶ After a type cast:
`int a = (int) doubleValue;`

Whitespace: Where NOT to Put Spaces

- ▶ Between a method name and an opening parenthesis (“(”):
`doIt(a)`

Whitespace: Where NOT to Put Spaces

- ▶ Between a method name and an opening parenthesis (“(”):
`doIt(a)`
- ▶ Between an unary operation (e.g., `++`, `--`) and its argument:
`a++`
`--b`

Whitespace: Where NOT to Put Spaces

- ▶ Between a method name and an opening parenthesis (“(”):
`doIt (a)`
- ▶ Between an unary operation (e.g., `++`, `--`) and its argument:
`a++`
`--b`
- ▶ Between a parenthesis (“(” or “)”) and its contents:
`(a + b)`

Whitespace: Where NOT to Put Spaces

- ▶ Between a method name and an opening parenthesis (“(”):
`doIt(a)`
- ▶ Between an unary operation (e.g., `++`, `--`) and its argument:
`a++`
`--b`
- ▶ Between a parenthesis (“(” or “)”) and its contents:
`(a + b)`
- ▶ Before brackets (“[” and “]”):
`int[] a`
`int[][] bb`
`a[1]`
`bb[1][2]`

Whitespace: Where to Put Newlines

- ▶ Before comments

Whitespace: Where to Put Newlines

- ▶ Before comments
- ▶ Between method declarations

Whitespace: Where to Put Newlines

- ▶ Before comments
- ▶ Between method declarations
- ▶ Between logical blocks inside a method

Whitespace: Where to Put Newlines

- ▶ Before comments
- ▶ Between method declarations
- ▶ Between logical blocks inside a method
- ▶ Between classes

Whitespace: Where to Put Newlines

- ▶ Before comments
- ▶ Between method declarations
- ▶ Between logical blocks inside a method
- ▶ Between classes
- ▶ Whenever the current line is too long

Whitespace: Where to Put Newlines

- ▶ Before comments
- ▶ Between method declarations
- ▶ Between logical blocks inside a method
- ▶ Between classes
- ▶ Whenever the current line is too long
 - ▶ indent the remainder of a wrapped line

Whitespace: Where to Put Newlines

- ▶ Before comments
- ▶ Between method declarations
- ▶ Between logical blocks inside a method
- ▶ Between classes
- ▶ Whenever the current line is too long
 - ▶ indent the remainder of a wrapped line
 - ▶ break lines after a comma

```
doIt (a,  
      b) ;
```

Whitespace: Where to Put Newlines

- ▶ Before comments
- ▶ Between method declarations
- ▶ Between logical blocks inside a method
- ▶ Between classes
- ▶ Whenever the current line is too long
 - ▶ indent the remainder of a wrapped line
 - ▶ break lines after a comma

```
doIt (a,  
      b) ;
```

- ▶ break lines before a binary operation

```
if ( (a > b)  
    && (b > c)  
    && (c > d) ) {
```


Blocks and Indentation

- ▶ Indentation width is 4 spaces.

Blocks and Indentation

- ▶ Indentation width is 4 spaces.
 - ▶ Or 1 TAB

Blocks and Indentation

- ▶ Indentation width is 4 spaces.
 - ▶ Or 1 TAB
 - ▶ **Never mix TABs and spaces**

Blocks and Indentation

- ▶ Indentation width is 4 spaces.
 - ▶ Or 1 TAB
 - ▶ **Never mix TABs and spaces**
- ▶ Bodies of control structures are always enclosed into curly braces (“{” and “}”)

Blocks and Indentation

- ▶ Indentation width is 4 spaces.
 - ▶ Or 1 TAB
 - ▶ **Never mix TABs and spaces**
- ▶ Bodies of control structures are always enclosed into curly braces (“{” and “}”)
 - ▶ **Even if there is only one line**

Blocks and Indentation

- ▶ Indentation width is 4 spaces.
 - ▶ Or 1 TAB
 - ▶ **Never mix TABs and spaces**
- ▶ Bodies of control structures are always enclosed into curly braces (“{” and “}”)
 - ▶ **Even if there is only one line**
- ▶ Block content must be indented:

```
if (a > b) {  
    a = 2 * a;  
    b--;  
}
```

Blocks and Indentation

- ▶ Indentation width is 4 spaces.
 - ▶ Or 1 TAB
 - ▶ **Never mix TABs and spaces**
- ▶ Bodies of control structures are always enclosed into curly braces (“{” and “}”)
 - ▶ **Even if there is only one line**
- ▶ Block content must be indented:

```
if (a > b) {  
    a = 2 * a;  
    b--;  
}
```

- ▶ Class content must be indented

```
class C {  
    private final int myValue = 0;  
    ...  
}
```

Conditional Operator

Single **if**:

```
if (condition) {  
  
}  
  
if (condition) {  
  
} else {  
  
}
```

“Cascade” **if**:

```
if (condition1) {  
  
} else if (condition2) {  
  
} else {  
  
}
```


When to Use the If Cascade

- ▶ It's needed to check several **related** conditions

When to Use the If Cascade

- ▶ It's needed to check several **related** conditions
- ▶ We cannot use **switch**:

```
if (a > b) {  
    ...  
} else if (a < b) {  
    ...  
} else if (a == b) {  
    ...  
}
```

Switch

```
switch (condition) {  
  case ABC:  
    statements;  
    /* falls through */  
  case DEF:  
    statements;  
    break;  
  case XYZ:  
    statements;  
    break;  
  default:  
    statements;  
    break;  
}
```

Loops

Precondition:

```
while (condition) {  
    // ...  
}
```

Postcondition:

```
do {  
    // ...  
} while (          );
```

For:

```
for (int i = 0; i < 10; i++) {  
    // ...  
}
```

Exception Handling

```
try {  
  
} catch (ExceptionClass e) {  
  
} finally {  
  
}
```

Exception Handling

```
try {  
  
} catch (ExceptionClass e) {  
  
} finally {  
  
}
```

- ▶ Never throw or catch

Exception Handling

```
try {  
  
} catch (ExceptionClass e) {  
  
} finally {  
  
}
```

▶ Never throw or catch

- ▶ `java.lang.NullPointerException`
- ▶ `java.lang.ClassCastException`
- ▶ `java.lang.RuntimeException`
- ▶ `java.lang.Exception`
- ▶ `java.lang.Throwable`

Exception Handling

```
try {  
  
} catch (ExceptionClass e) {  
  
} finally {  
  
}
```

- ▶ Never throw or catch

- ▶ `java.lang.NullPointerException`
- ▶ `java.lang.ClassCastException`
- ▶ `java.lang.RuntimeException`
- ▶ `java.lang.Exception`
- ▶ `java.lang.Throwable`

- ▶ Never catch `java.lang.Error`

Idiots

Idioms

Immediately Returning Ifs

Bad

```
if (cond) {  
    return true;  
} else {  
    return false;  
}
```

Immediately Returning Ifs

Bad

```
if (cond) {  
    return true;  
} else {  
    return false;  
}
```

Good

```
return cond;
```

Not Recommended

```
if (cond) {  
    ...  
    ...  
    return;  
} else {  
    doIt();  
}
```

Redundant Else

Not Recommended

```
if (cond) {  
    ...  
    ...  
    return;  
} else {  
    doIt();  
}
```

Recommended

```
if (cond) {  
    ...  
    ...  
    return;  
}  
doIt();
```

Do NOT Use Expressions with Side-Effects

Bad

```
int a = b = c = 0;
```

Very Bad

```
if (a = b) {  
    ...  
}
```

Bad

```
int a = b[c++];
```

Why We Need Conventions

- ▶ Code is complex

Why We Need Conventions

- ▶ Code is complex
- ▶ Code lives long

Why We Need Conventions

- ▶ Code is complex
- ▶ Code lives long
- ▶ Code is written and read by many people

Why We Need Conventions

- ▶ Code is complex
- ▶ Code lives long
- ▶ Code is written and read by many people
- ▶ Code is almost never properly documented

Why We Need Conventions

- ▶ Code is complex
- ▶ Code lives long
- ▶ Code is written and read by many people
- ▶ Code is almost never properly documented

Why We Need Conventions

- ▶ Code is complex
- ▶ Code lives long
- ▶ Code is written and read by many people
- ▶ Code is almost never properly documented

NB: Some additional reasons are mentioned in CCJ

If You Remember Only One Thing ...

Always look for explanations!

The STRONGLY Recommended Book

