

# Topic 5: Control Flow

Programming Practice and Applications (4CCS1PPA)

---

Dr. Martin Chapman  
Thursday 20th October

programming@kcl.ac.uk  
martinchapman.co.uk/teaching

To understand the different (additional) ways in which we can alter the **order** in which our program is executed:

- **Conditional** statements (If statements)
- **Iterative** statements (Loops)
- **Recursion**

# Conditional Statements

---

## REMEMBER: ENCAPSULATION (1)

Previously, we identified that by keeping the balance field in our bank account **private**, we were correctly **encapsulating** our code.

- We only allow users to interact with a set **interface** (e.g. deposit and withdraw), rather than allowing them to directly manipulate the internal **state** of an account (i.e. the balance).

But this was not enough, and we needed some more **syntax** to take full **advantage** of encapsulation.

# VALIDATING PARAMETERS INSIDE A METHOD

```
public class BankAccount {
```

```
    private double balance;
```

```
    public void withdraw(double amount, int pin) {
```

*Ensure pin is correct before:*

```
        balance = balance - amount;
```

*If not, print:*

```
        System.out.println("Ah ah ah, you didn't say the magic word.");
```

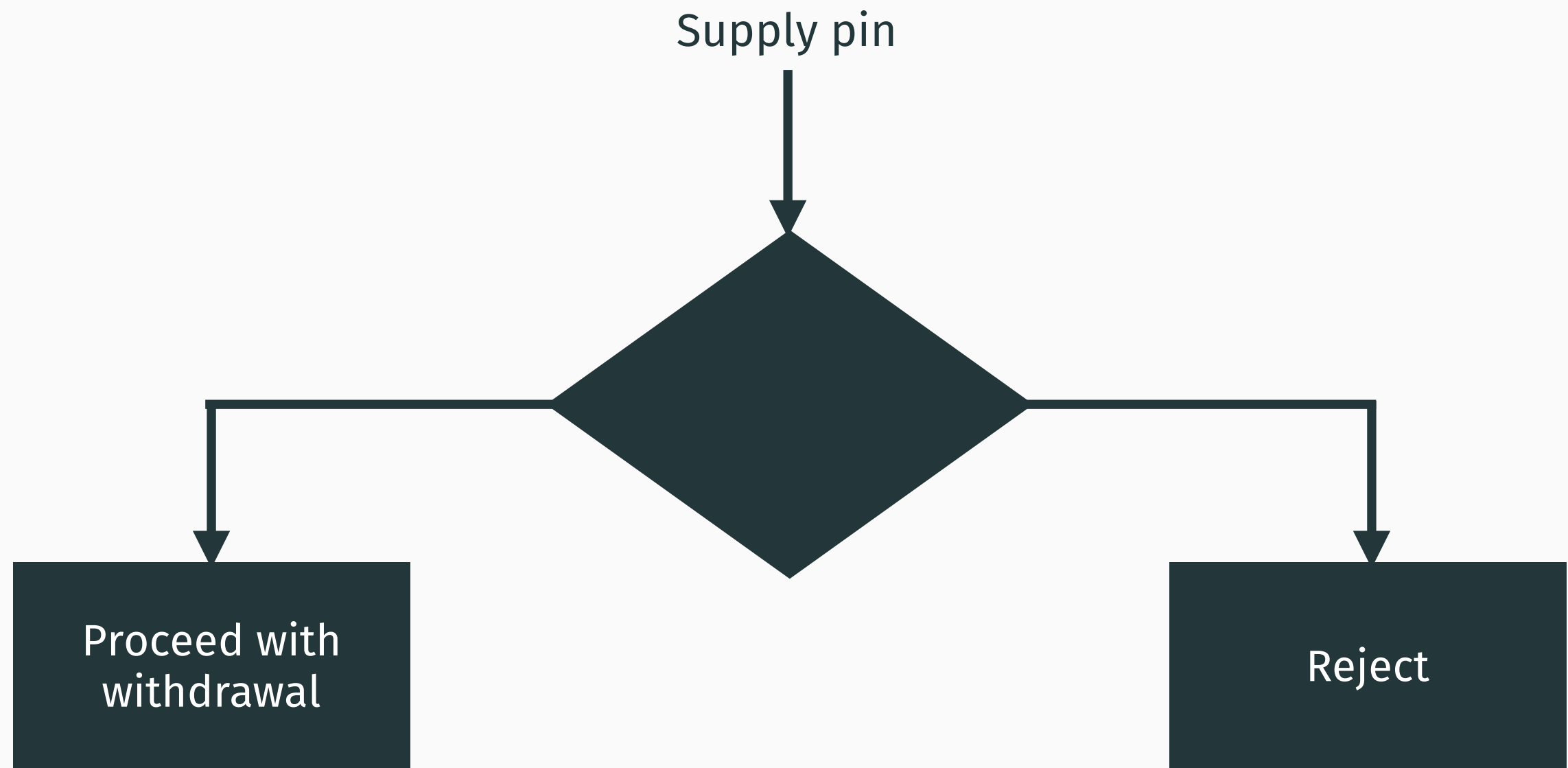
```
}
```



MakeAGIF.com

# DECISIONS

Specifically, we need our program to make a **decision**, based upon the **pin** supplied to the withdraw method, about whether to proceed with the withdrawal or to reject the withdrawal.



```
balance = balance - amount;
```

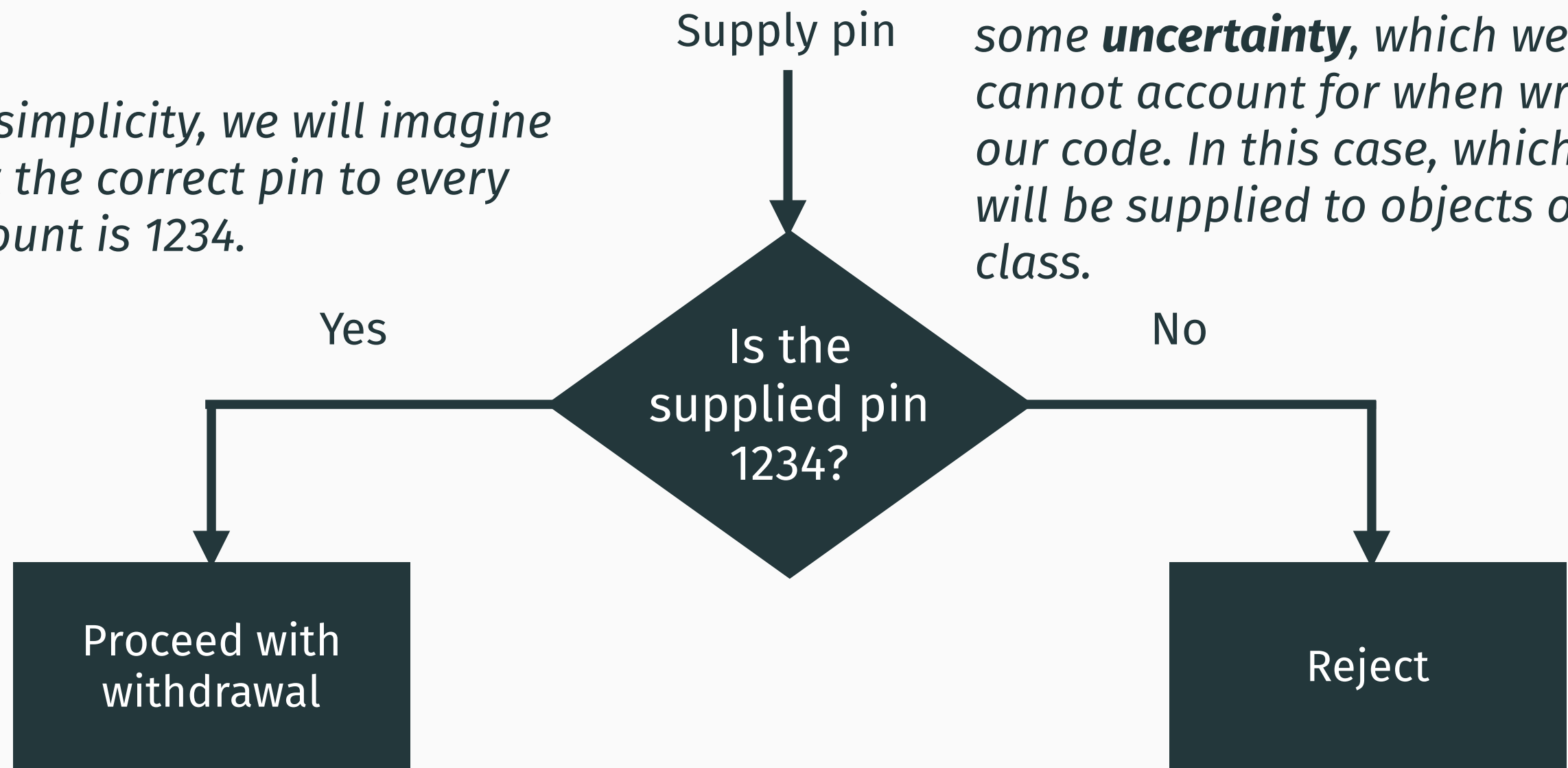
```
System.out.println("Ah ah ah
```

# FROM DECISIONS TO QUESTIONS

In order to make a decision, we need to ask **at least one question** with a **yes** or a **no** answer. This answer guides our program down a **distinct branch of execution**.

*For simplicity, we will imagine that the correct pin to every account is 1234.*

*We ask questions because of some **uncertainty**, which we cannot account for when writing our code. In this case, which pin will be supplied to objects of our class.*

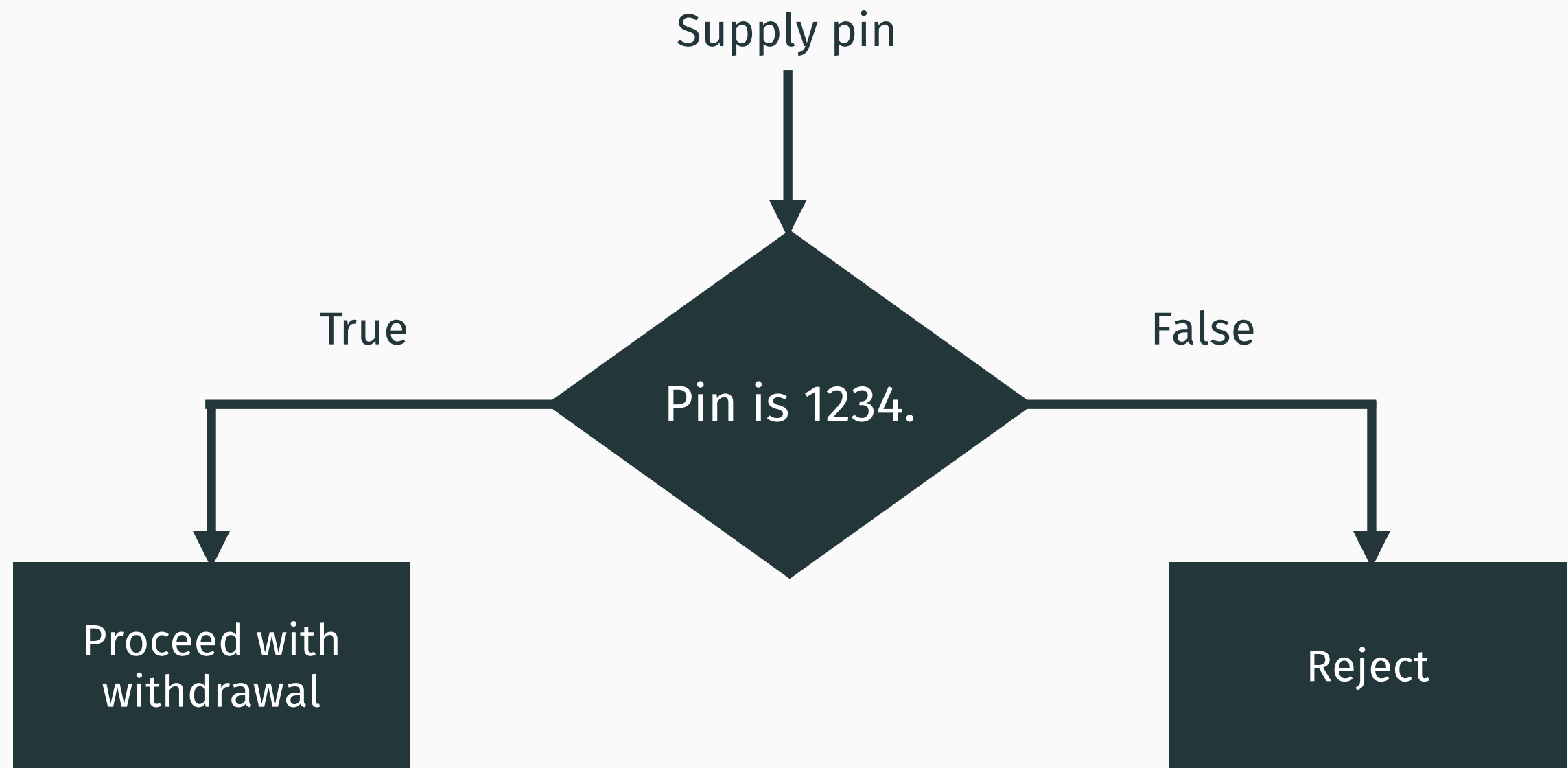


```
balance = balance - amount;
```

```
System.out.println("Ah ah ah
```

# FROM QUESTIONS TO CONDITIONS

To be even more precise still, we need to phrase our question as a **condition** (a state in the program), which is either **true** or **false**.



```
balance = balance - amount;
```

```
System.out.println("Ah ah ah
```



We make utterances every day containing conditions that are either true or false:

- It is ok to make students sit for three hours once a week in order to learn programming. **False.**
- Academics receive generous salaries. **False.**
- Martin should be immediately promoted to Professor. **True.**

# GENERATING BOOLEAN VALUES

During your investigation of different primitive types (Topic2-2), you should have experimented with the **boolean** type.

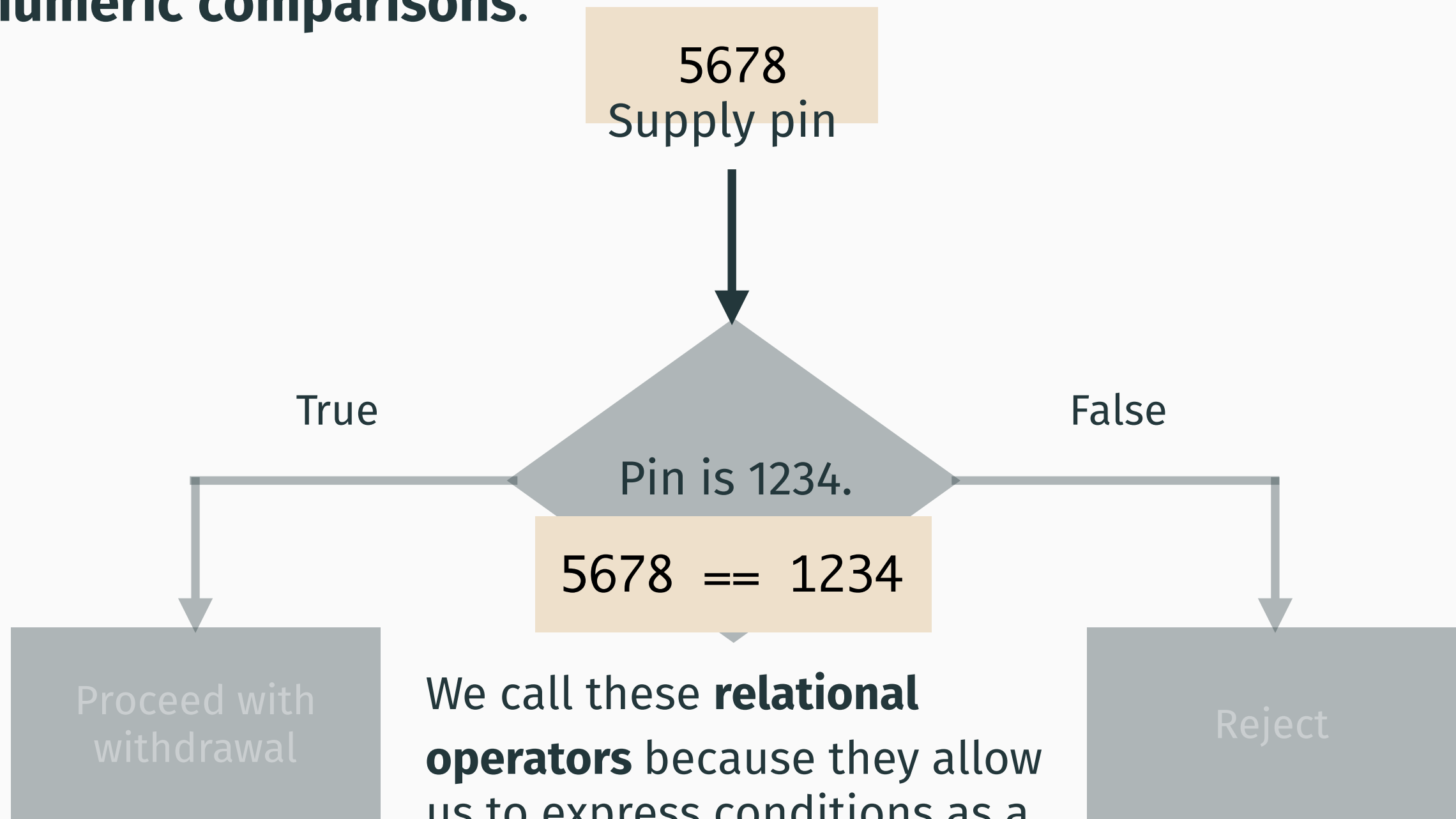
The values that can be placed inside variables of a boolean type are **true** and **false**.

Thus, for something to be used as a condition, it must **generate** a boolean value when **evaluated** (e.g. when it is printed).

There are a number of different ways to generate boolean values.

# GENERATING BOOLEAN VALUES (1): NUMERIC COMPARISONS (1)

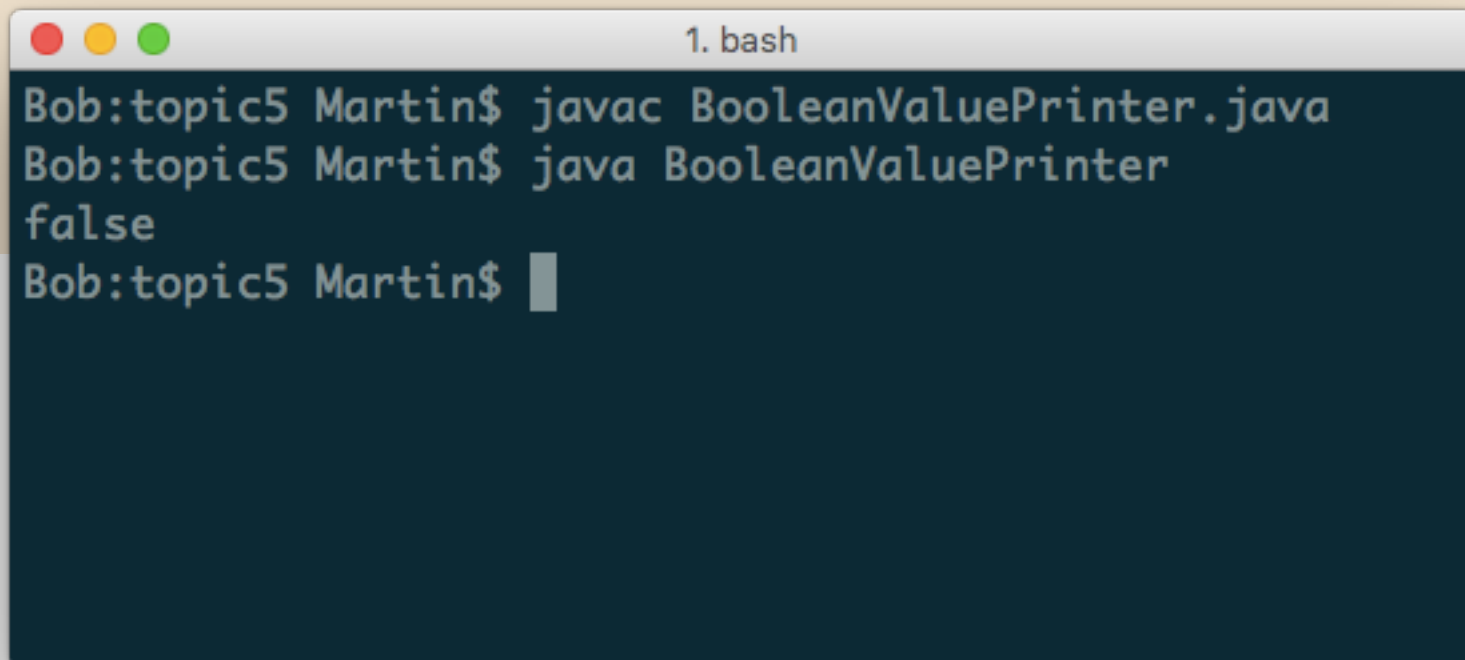
The first way to generate boolean values is by making **numeric comparisons**.



We call these **relational operators** because they allow us to express conditions as a **relationship** between two entities.

# GENERATING BOOLEAN VALUES (1): NUMERIC COMPARISONS (2)

```
public class BooleanValuePrinter {  
    public static void main(String[] args) {  
        System.out.println(5678 == 1234);  
    }  
}
```



```
1. bash  
Bob:topic5 Martin$ javac BooleanValuePrinter.java  
Bob:topic5 Martin$ java BooleanValuePrinter  
false  
Bob:topic5 Martin$
```

# GENERATING BOOLEAN VALUES (1): NUMERIC COMPARISONS (3)

There are other relational operators available to us to make numeric comparisons, and thus generate boolean values.

Java	Mathematical Notation	Description
>	$>$	Greater than
>=	$\geq$	Greater than or equal
<	$<$	Less than
<=	$\leq$	Less than or equal
==	$=$	Equal
!=	$\neq$	Not equal

## ASIDE: EQUALS VS. DOUBLE EQUALS

So far we have seen both equals and double equals.

- Equals is used for **assignment**.
- Double equals is used for **comparison**.

It's very common for programmers to mix the two up.

Currently, it's easy for the compiler to catch this mistake.

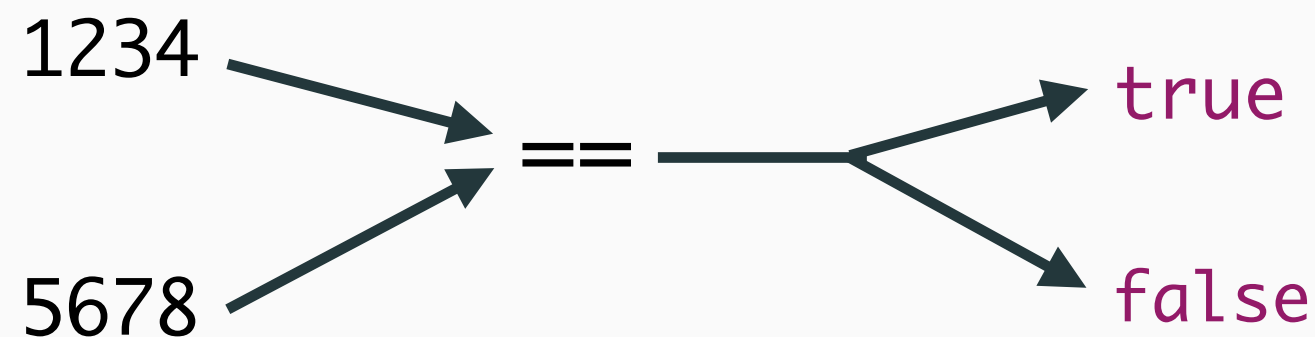
```
System.out.println(5678 = 1234);
```

But in other circumstances this mistake may cause unexpected behaviour.

```
int fiveSixSevenEight = 5678;  
System.out.println(fiveSixSevenEight = 1234);
```

# RELATIONAL OPERATORS AS METHODS

It might be helpful to view relational operators as a form of very simple method, which (typically) takes two numbers as input, performs a process, and then gives **true** or **false** as output.



In reality there are specific bytecode instructions for these operations, and an operand stack.

**Open question:** Do relational operators only operate on numbers (or variables containing numbers)?



In the laboratory, use the template provided by the `BooleanValuePrinter` class to experiment with generating different boolean values from numeric comparisons.

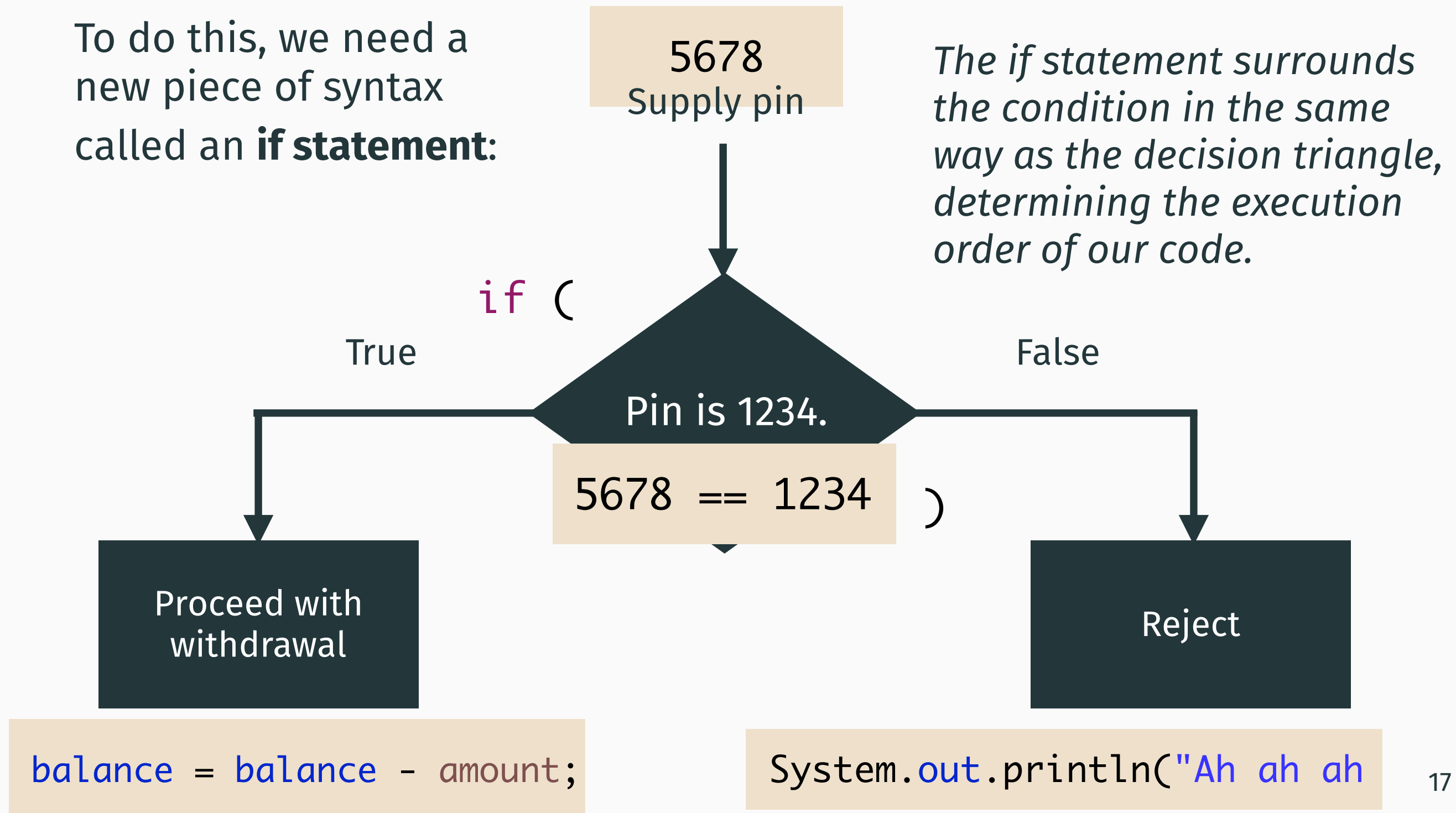
- Try out all the different operators listed on Slide 13.



# TESTING BOOLEAN VALUES (1)

But of course we don't just want to generate boolean values from conditions, we also want to **test** the resulting value, and thus alter the execution order of our program depending on the result.

To do this, we need a new piece of syntax called an **if statement**:



# CONDITIONAL (IF) STATEMENTS (1)

Pin is 1234.

*We can view the braces as being equivalent to our arrows in the previous diagram, directing the flow of execution.*

```
if ( 5678 == 1234 ) ← True
    balance = balance - amount;
} else {    False

    System.out.println("Ah ah ah you didn't say the magic word.");
}
```

Proceed with  
withdrawal

Reject

When we write an if statement, sections of our code are executed while other sections of our code are **not**, depending on the boolean value generated by a condition (in this example, this value is always false).

Like **calling a method**, this contradicts the typical **line-by-line** execution of a program.

We make decisions based upon the boolean value of conditions every day.

- **Condition:** It is a hot day today (`day equals hot')
- **If true:** Throw on shorts and pink flip-flops.
- **If false:** Throw on bright yellow jumper.

# REMEMBER: ENCAPSULATION (3)

We can now put in place the necessary logic to complement our encapsulated bank account class.

```
public void withdraw(double amount, int pin) {
```

```
    if (pin == 1234 ) {  
        balance = balance - amount;
```

```
    } else {
```

```
        System.out.println("Ah ah ah, you didn't say the magic word.");
```

```
    }  
}
```

*If statements give us yet more blocks of code in a program.*

*Because variables contain values, it makes sense that we can replace values in conditions (and thus to relational operators) with variables. This allows us to **test unknown values**.*

*Once one of the conditional blocks finishes, the JVM starts to execute any code below the last block.*

Different sections of our code are executed depending on the values passed to the method.



# REMEMBER: INITIAL DEPOSIT (OPTIONAL ELSE CLAUSE)

```
public class BankAccount {  
    private double balance;  
  
    public BankAccount(int initialDeposit) {  
        if ( initialDeposit > 0 ) {  
            balance = initialDeposit;  
        }  
    }  
}
```

*As we can see here, the else clause  
is optional.*

We can also now avoid the issue of users making erroneous initial deposits.

## ASIDE: OMITTING THE BRACES (1)

```
public class BankAccount {  
    private double balance;  
  
    public BankAccount(int initialDeposit) {  
        if ( initialDeposit > 0 ) balance = initialDeposit;  
    }  
}
```

When we don't use an else statement, we can, if we choose to, **omit** the braces from our if statement.

This can produce **cleaner** code.

## ASIDE: OMITTING THE BRACES (2)

There are certain issues with omitting the braces from an if statement:

- Only **expression statements** (statements that do not contain variable **declarations** or **control flow** statements) can follow a conditional without braces.
- There is a greater potential for error.

```
if ( initialDeposit > 0 )  
    balance = initialDeposit;  
    System.out.println("Balance deposited");
```

*This may look like both statements are only executed if the deposit is valid, but the print statement is actually **always** executed.*

I personally quite like this shorthand (lots of people do not), but I use it **carefully**.

# LECTURE EXERCISE: BOOK (1)

Let's try and apply these ideas in a **different** context.

Let's model a book.

As before, let's ask the question what's the **state**

- Current page
- Total pages

and **behaviour** of a book?

- Read a page
- Go back a page
- Read page number

*Behaviour isn't always exhibited by an object, it may also be performed **on** that object.*



## LECTURE EXERCISE: BOOK (2), FIELDS

```
public class Book {  
  
    private int currentPage;  
    private int totalPages;  
  
    public Book(int totalPages) {  
  
        currentPage = 1;  
        this.totalPages = totalPages;  
  
    }  
  
}
```

*We'll use a constructor to force a user to supply the total number of pages (because the logic of our code will rely on this value), and to set the default value for the current page when the book is first created.*

# LECTURE EXERCISE: BOOK (3), READ PAGE

```
public void readPage() {  
    if ( currentPage < totalPages ) {  
        currentPage = currentPage + 1;  
    }  
}
```

*It makes sense that we can only keep reading pages while there are pages to read!*

## LECTURE EXERCISE: BOOK (4), TURN PAGE BACK

```
public void turnPageBack() {  
    if ( currentPage > 1 ) {  
        currentPage = currentPage - 1;  
    }  
}
```

*Similarly, it makes sense that we can only turn back as far as the first page of the book.*

## LECTURE EXERCISE: BOOK (5), READ PAGE NUMBER

```
public int readPageNumber() {  
    return currentPage;  
}
```

*We add a simple accessor method to determine the current page.*

# LECTURE EXERCISE: BOOK (6)

```
public class Book {  
  
    private int currentPage;  
    private int totalPages;  
  
    public Book(int totalPages) {  
        currentPage = 1;  
        this.totalPages = totalPages;  
    }  
  
    public void readPage() {  
        if ( currentPage < totalPages ) {  
            currentPage = currentPage + 1;  
        }  
    }  
}
```

```
    public void turnPageBack() {  
        if ( currentPage > 1 ) {  
            currentPage = currentPage - 1;  
        }  
    }  
  
    public int readPageNumber() {  
        return currentPage;  
    }  
}
```

# LECTURE EXERCISE: BOOK (7), READING A BOOK

*Our Driver class does not always have to literally be called driver.*

```
public class BookReader {  
  
    public static void main(String[] args) {  
  
        Book aGameOfThrones = new Book(704);  
        aGameOfThrones.readPage();  
        aGameOfThrones.readPage();  
        aGameOfThrones.readPage();  
        System.out.println(aGameOfThrones.readPageNumber());  
        aGameOfThrones.turnPageBack();  
        System.out.println(aGameOfThrones.readPageNumber());  
  
    }  
}
```

*Note how we can also encode some information about the book using the object name.*

## BACK TO OUR BANK ACCOUNT: BEEFING UP SECURITY (1)

Let's imagine that we want to secure our account even further, by asking those withdrawing money to supply **two** pin numbers (think two factor authentication), rather than just one.

How would this affect our conditional statement?

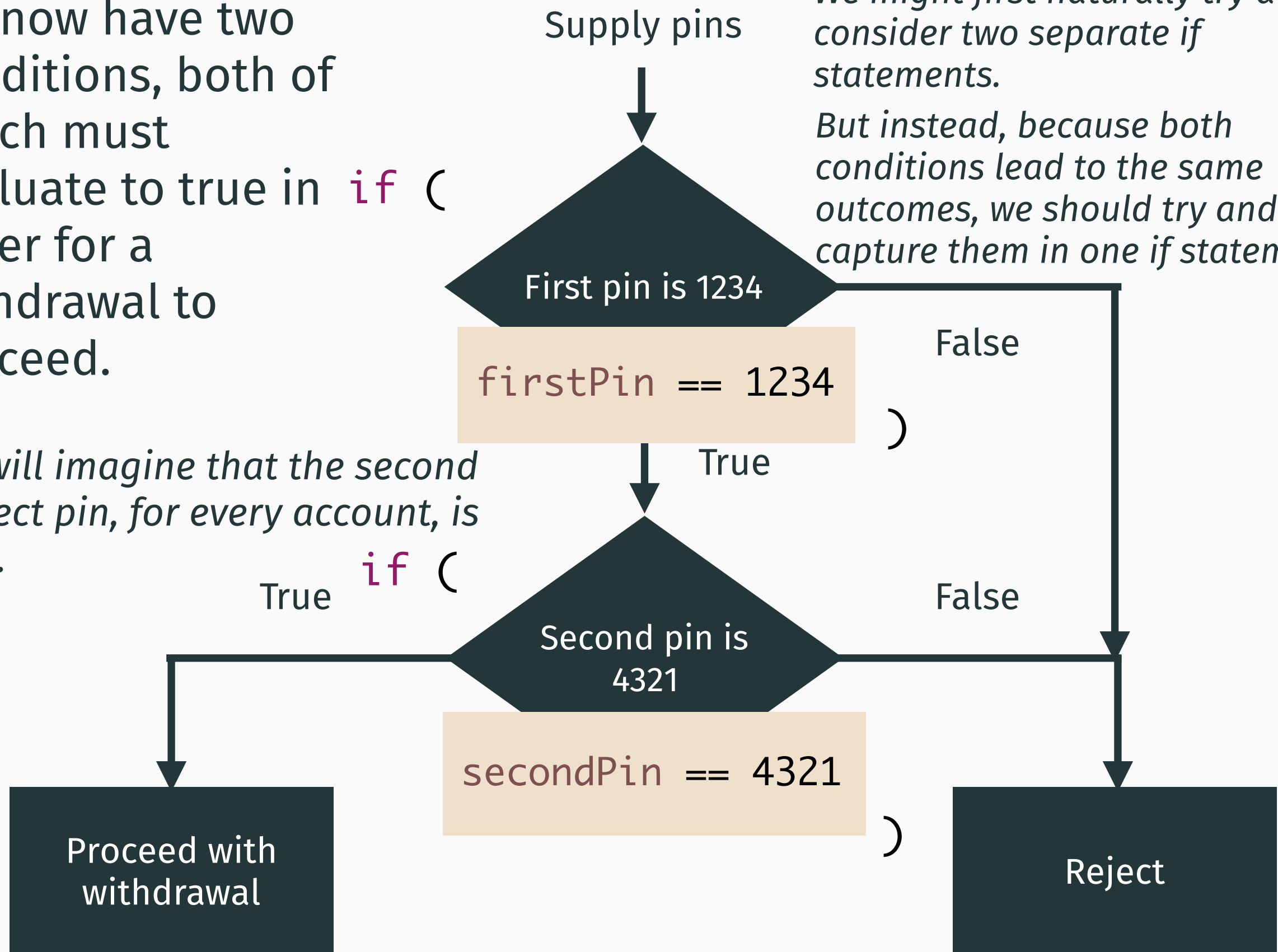
```
public void withdraw(double amount, int firstPin, int secondPin) {  
    if ( firstPin == 1234 ) {  
        balance = balance - amount;  
    } else {  
        System.out.println("Ah ah ah, you didn't say the magic word.");  
    }  
}
```

# TESTING BOOLEAN VALUES (2)

We now have two conditions, both of which must evaluate to true in `if` (order for a withdrawal to proceed.

We will imagine that the second correct pin, for every account, is 4321.

*We might first naturally try and consider two separate if statements. But instead, because both conditions lead to the same outcomes, we should try and capture them in one if statement.*





# BANK ACCOUNT: BEEFING UP SECURITY (2)

By the same process as before, we formulate the following code:

First pin is 1234

Second pin is  
4321

*How do we connect these two conditions?*

```
public void withdraw(double amount, int firstPin, int secondPin) {  
    if ( firstPin == 1234  secondPin == 4321 ) {  
        balance = balance - amount;  
    } else {  
        System.out.println("Ah ah ah, you didn't say the magic word.");  
    }  
}
```

Another way to generate boolean values, and thus construct conditions, is to **combine** existing boolean values.

- These values may themselves have been generated by other conditions (to an arbitrary level of nesting).

This is an idea you should be familiar with from **logic**.

# GENERATING BOOLEAN VALUES (2): OTHER BOOLEAN VALUES (2)

Conditional operators:

**&&**

**A      B      A AND B**

TRUE TRUE TRUE

TRUE FALSE FALSE

FALSE TRUE FALSE

FALSE TRUE FALSE

**||**

**A      B      A OR B**

TRUE TRUE TRUE

TRUE FALSE TRUE

FALSE TRUE TRUE

FALSE FALSE FALSE

Unary operator:

**!**

**A      NOT A**

TRUE FALSE

FALSE TRUE

*We won't consider bitwise operations or shifts in this course.*

We make decisions based upon the boolean value of (combined) conditions every day.

- **Condition:** It is a hot day today **OR** I love pink.
- **If true:** Throw on shorts and pink flip-flops.
- **If false:** Throw on bright yellow jumper.

## CONDITIONAL (IF) STATEMENTS (2)

In the case of our bank account, because we need **both** of the pins to be correct, we select a logical **and**.

```
public void withdraw(double amount, int firstPin, int secondPin) {  
    if ( firstPin == 1234 && secondPin == 4321 ) {  
        balance = balance - amount;  
    } else {  
        System.out.println("Ah ah ah, you didn't say the magic word.");  
    }  
}
```

*Here we generate a boolean value on the left and the right, from separate conditions, and then combine them into a **subsequent** boolean value.*

# CONDITIONAL (IF) STATEMENTS (3)

Again, depending on the values passed to our method, different sections of our code are executed.

```
public void withdraw(double amount, int 1234 , int 5678 ) {  
    if ( false ) { ←  
        balance = balance - amount;  
    } else {  
        System.out.println("Ah ah ah, you didn't say the magic word.");  
    }  
}
```



# DIFFERENT SECURITY MECHANISMS

In the laboratory, implement a **variable pin system**, such that each account has a **different correct pin**.

Then, try out **different** security mechanisms. Here are some (not strictly logical) ideas:

- **Either** pin can be correct.
- **Only one** pin should be correct.
- One pin **cannot be equal to** a certain value (a honeypot?).

## ASIDE: PRECEDENCE WITH MULTIPLE OPERATORS (1)

Now that we have the freedom to use different operators in our program, a natural question to ask is in which **order** does Java evaluate the different operators that we have seen so far?

```
public class OperatorPrecedence {  
    public static void main(String[] args) {  
        System.out.println( 1 + 1 == 2 || 3 - 1 == 3 &&  
                             5 == 5 && 7 > 4 && !false );  
    }  
}
```

*I've used multiple lines here for space.*



## ASIDE: PRECEDENCE WITH MULTIPLE OPERATORS (2)

```
1 + 1 == 2 || 3 - 1 == 3 && 5 == 5 && 7 > 4 && !false
```

Negation (unary):

```
1 + 1 == 2 || 3 - 1 == 3 && 5 == 5 && 7 > 4 && true
```

Addition:

```
2 == 2 || 3 - 1 == 3 && 5 == 5 && 7 > 4 && true
```

Subtraction (left to right because of **equal** precedence.):

```
2 == 2 || 2 == 3 && 5 == 5 && 7 > 4 && true
```

(Non-equality) Relational Operators:

```
2 == 2 || 2 == 3 && 5 == 5 && true && true
```

## ASIDE: PRECEDENCE WITH MULTIPLE OPERATORS (3)

```
2 == 2 || 2 == 3 && 5 == 5 && true && true
```

Equality:

```
true || false && true && true && true
```

Logical And:

```
true || false
```

Logical Or:

```
true
```

We'll update our notion of operator precedence once we've looked at arithmetic in more detail.



## ASIDE: PRECEDENCE WITH MULTIPLE OPERATORS (4): BRACKETS

Like regular mathematics, we can change the order in which statements are evaluated, and thus the resulting boolean value, using **brackets**.

```
true || false && false
```

```
true
```

```
( true || false ) && false
```

```
false
```

Always be conscious, in your own code, of the natural precedence of operators, and whether you need to adjust this to have a condition evaluated in the correct way.

I tend to use brackets to increase **readability**, even if the natural ordering of the operators involved is fine.

- We will hopefully see examples of this going forward.



Let's return to our book class, and update it slightly.

Let's remove the **obligation** to supply a maximum page limit.

- Perhaps the book has no ending (George R.R. Martin...)

This **requirement** will necessitate two changes:

- The way in which the book object is constructed.
- The way in which we turn a page.

# LECTURE EXERCISE: BOOK (9), ADDITIONAL CONSTRUCTOR

```
private int currentPage;  
private int totalPages;  
  
public Book(int totalPages) {  
  
    currentPage = 1;  
    this.totalPages = totalPages;  
  
}  
  
public Book() {  
  
    this(0);  
  
}
```

*Remember multiple constructors give us different patterns we can match when we create an object of a class.*

# LECTURE EXERCISE: BOOK (10), READ PAGE LOGIC

```
public void readPage() {  
    if ( totalPages == 0  || currentPage < totalPages ) {  
        currentPage = currentPage + 1;  
    }  
}
```

*If there's no maximum page, we can simply permit page turns for as long as the user wishes.*

# BACK TO BANK ACCOUNT: NO OVERDRAFT (1)

```
public void withdraw(double amount, int firstPin, int secondPin) {  
    if ( firstPin == 1234 && secondPin == 4321 ) {  
        balance = balance - amount;  
    } else {  
        System.out.println("Ah ah ah, you didn't say the magic word.");  
    }  
}
```

`balance - amount >= 0`

What if we want to add an additional condition somewhere in our program, that ensures we are only able to withdraw an amount from our account if the result of that subtraction **isn't negative**.

- In other words, we're denying our user an **overdraft** (sounds like my bank account).

## BACK TO BANK ACCOUNT: NO OVERDRAFT (2)

```
public void withdraw(double amount, int firstPin, int secondPin) {  
    if ( firstPin == 1234 && secondPin == 4321 && balance - amount >= 0 ) {  
        balance = balance - amount;  
    } else {  
        System.out.println("                ");  
    }  
}
```

As there's no **reassignment** here, we can **test** the result of an arithmetic operation **without affecting the value** of the operands involved.

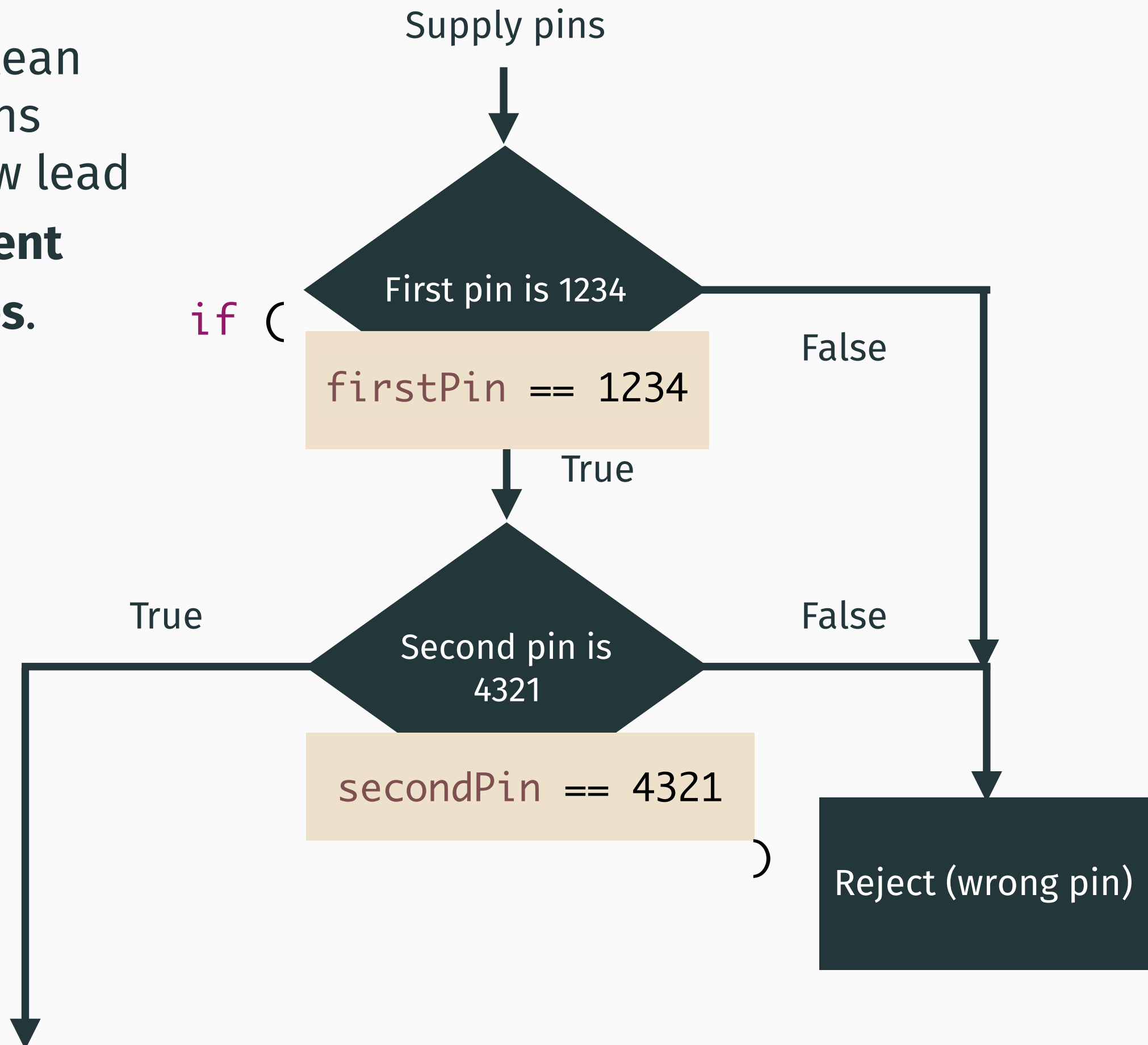
We might try and add this condition into our existing if statement, but then how do we construct an appropriate error message?

- It would **confuse** the user of our account, if they didn't realise the balance restriction, but entered both their pins in correctly.

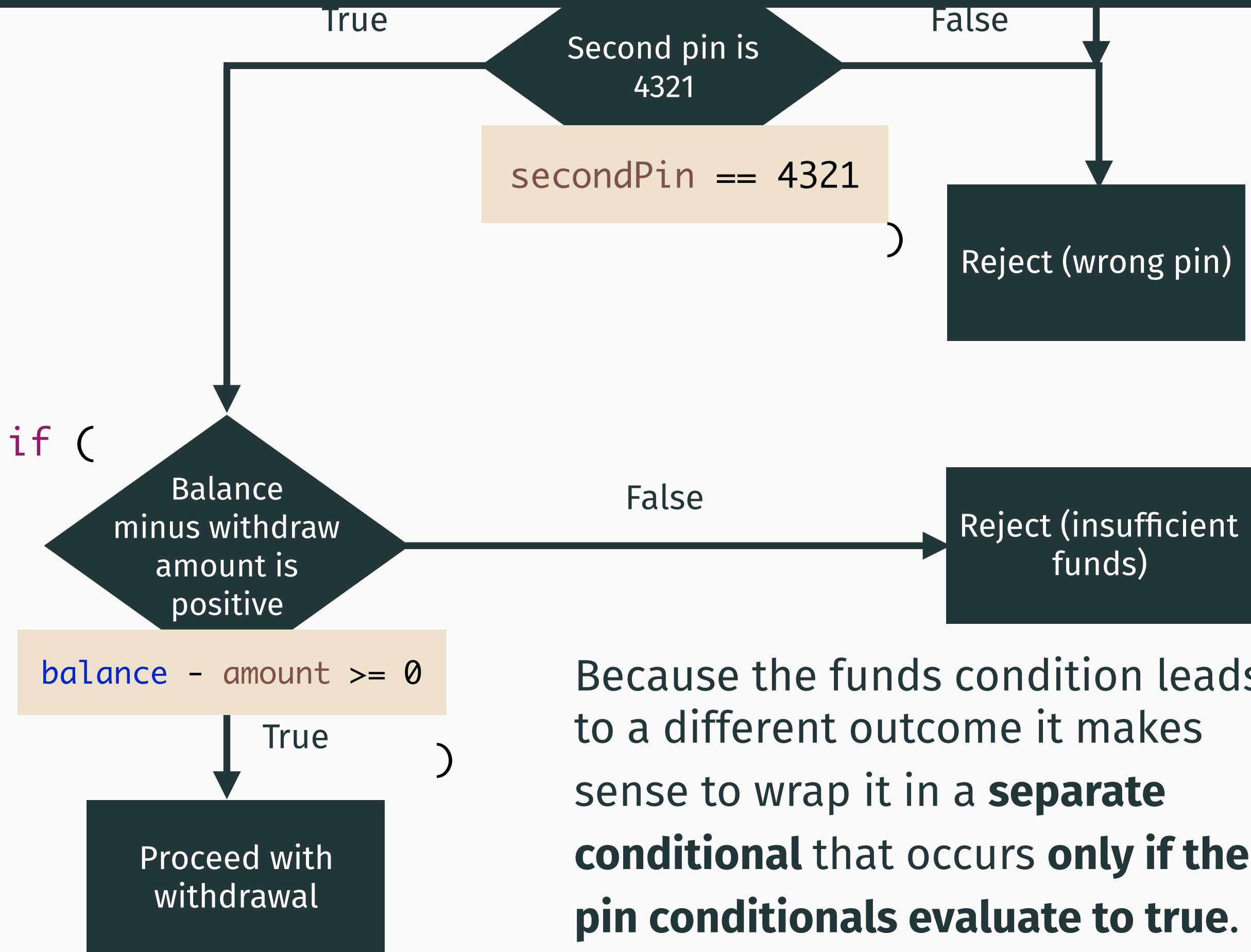


# TESTING BOOLEAN VALUES (3)

Our boolean conditions must now lead to **different outcomes**.



# TESTING BOOLEAN VALUES (3)



Because the funds condition leads to a different outcome it makes sense to wrap it in a **separate conditional** that occurs **only if the pin conditionals evaluate to true.**

# NESTED IF STATEMENTS (1)

First pin is 1234

Second pin is  
4321

```
if ( firstPin == 1234 && secondPin == 4321 ) {
```

```
    if ( balance - amount >= 0 ) {
```

```
        balance = balance - amount;
```

```
    } else {
```

```
        System.out.println("Insufficient funds.");
```

```
    }
```

```
} else {
```

```
    System.out.println("Ah ah ah, you didn't say the magic word.");
```

```
}
```

Balance  
minus withdraw  
amount is  
positive

*We then have the option to show  
two distinct error messages.*

## NESTED IF STATEMENTS (2)

When a conditional can only be evaluated in the event that a previous conditional has been evaluated (i.e. it appears inside one of the braces of another conditional), then we call that conditional a **nested if statement**.

```
if ( firstPin == 1234 && secondPin == 4321 ) {  
    if ( balance - amount >= 0 ) {  
        balance = balance - amount;  
    } else {  
        System.out.println("Insufficient funds.");  
    }  
} else {  
    System.out.println("Ah ah ah, you didn't say the magic word.");  
}
```

## ASIDE: OMITTING THE BRACES (3)

Once we have nested if statements, the dangers of omitting the braces in a conditional statement become clearer:

```
if ( firstPin == 1234 && secondPin == 4321 )
```

```
    if ( balance - amount > 0 )
```

```
        balance = balance - amount;
```

```
else
```

```
    System.out.println("Ah ah ah, you didn't say the magic word.");
```

*This isn't particularly readable.*

*And if we were to remove our inner else, this code may exhibit unexpected behaviour.*

*Although it isn't indented, this else is now **associated with the inner (closest) if statement.***

## ASIDE: NESTED IFS VS. LOGICAL AND

Given that we know a nested if is only executed in the presence of a previous **parent** if, we could reformat those conditionals with a **logical and** as nested ifs.

But, in most cases, this reduces readability and may result in code **repetition**.

```
if ( firstPin == 1234 ) {  
    if ( secondPin == 4321 ) {  
        ...  
    } else {  
        System.out.println("Ah ah ah, you didn't say the magic word.");  
    }  
} else {  
    System.out.println("Ah ah ah, you didn't say the magic word.");  
}
```

# BANK ACCOUNT: PERMITTING AN OVERDRAFT

Back in our bank account, let's imagine that we now do permit those customers with an account the ability to obtain an overdraft, but want to charge them (a simple form of) **interest** in doing so.

- If customers don't go below zero with a withdrawal, they aren't charged any interest.

```
balance = balance - amount;
```

- **Otherwise** if customers go below £100 they are charged a one off **£20** fee.

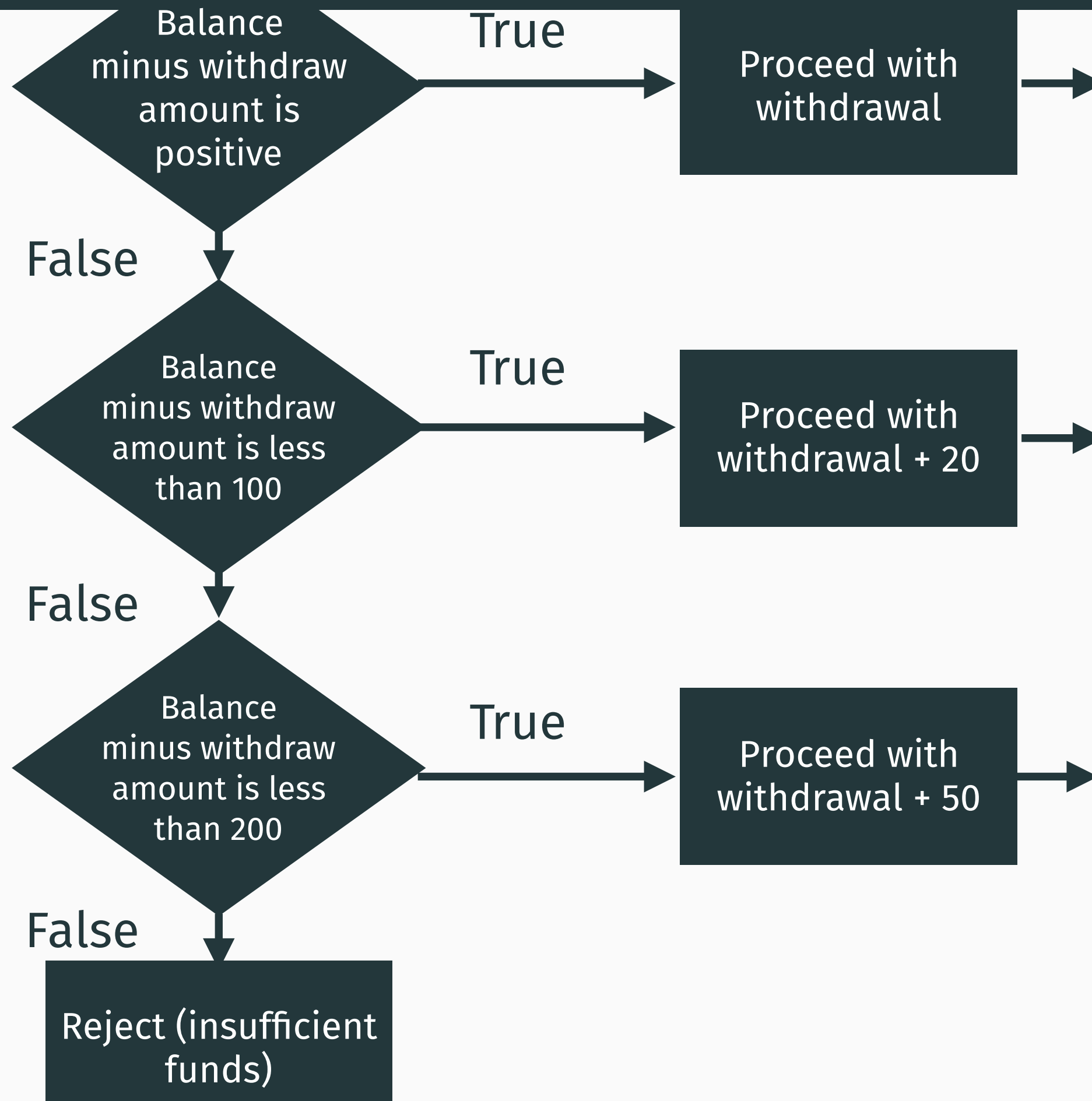
```
balance = balance - ( amount + 20 );
```

*Note the  
brackets  
here*

- **Otherwise** if customers go below £200 they are charged a one off **£50** fee.

```
balance = balance - ( amount + 50 );
```

# TESTING BOOLEAN VALUES (3)





# MODELLING MULTIPLE ALTERNATIVES (1)

How do we approach this problem given the syntax we've seen so far?

- We can't use **operators**, because this implies that two conditions will exist within the same if statement, but each condition stated previously has a **different outcome**.
- We can't use **nested if statements**, because none of the conditions depend on the prior conditions being fulfilled.
- The most **natural** thing we can do at this stage is probably the following:

# MODELLING MULTIPLE ALTERNATIVES (2)

```
if ( 0 - 200 >= 0 ) { ←
    balance = balance - amount;
} else {
    System.out.println("Insufficient funds.");
}

if ( 0 - 200 <= -100 ) {
    -220 = 0 - ( 200 + 20 );
}

if ( -220 - 200 <= -200 ) {
    -470 = -220 - ( 200 + 50 );
}
```

What's wrong with this?

If someone has a balance of £0 and withdraws £200, this withdrawal will happen **twice**, and they will be charged an additional £20 in fees (not to mention the misleading error message).

We say that separated conditionals like this **aren't mutually exclusive**.

# THE ELSE IF STATEMENT

*Note how  
order is  
important.*

```
if ( balance - amount > 0 ) {  
    balance = balance - amount;  
}  
else if ( balance - amount <= -200 ) {  
    balance = balance - ( amount + 50 );  
}  
else if ( balance - amount <= -100 ) {  
    balance = balance - ( amount + 20 );  
}  
else {  
    System.out.println(" ");  
}
```

*The else block is still optional, but can be used to specify **default** behaviour if none of the conditions above are satisfied.*

In order to have the mutually exclusivity we need, we have to group our conditionals in a set of **connected blocks** using a new piece of syntax called an **else if** statement.

Only **the first block with a satisfied conditional will execute.**

## ASIDE: RETURN FROM VOID METHODS

```
public void withdraw(double amount) {  
    if ( balance - amount > 0 ) {  
        balance = balance - amount;  
        return;  
    }  
  
    if ( balance - amount < -200 ) {  
        balance = balance - ( amount + 50 );  
        return;  
    }  
  
    . . .  
}
```

*Even if a method is void, we can still call the return command in order to **exit** that method **early**.*

There is something we could do to address the mutual exclusivity issue here.

We could **return** from our method within each if statement, and thus not complete any of the code following that if statement if it is satisfied.

Using multiple returns for control flow is **contentious** (more later).

# SUMMARY: CONDITIONAL CONSTRUCTS

If we want to execute different parts of our code based on different conditions, we have the following constructs available to us:

- **If-else blocks** When we want to test a single condition, or logically combined conditions, with a distinct true-false outcome.
- **Nested if blocks** When we to test a series of conditions, with potentially different true-false outcomes.
- **If-elseif-else blocks** When we want to select one outcome from a set of outcomes, each of which is predicated on an individual (logically combined) condition.

# ALTERNATIVE CONDITIONAL SYNTAX (1): TERNARY OPERATOR (1)

We have seen unary and binary operators, but there is also a (potentially useful) **ternary** operator available to us, that allows us to express **single line if-else statements**.

We could rephrase the following...

```
if ( balance - amount > 0 ) {  
    balance = balance - amount;  
} else {  
    System.out.println("Insufficient funds.");  
}
```

...as...

```
balance = balance - amount >= 0 ? balance - amount : balance;
```

We call this the **ternary operator**.



# ALTERNATIVE CONDITIONAL SYNTAX (1): TERNARY OPERATOR (2)

We must start with an assignment.

The value to assign if the boolean value generated is false.

We separate each part of the operator with an equals, a question mark and then a colon, respectively.

```
balance = balance - amount >= 0 ? balance - amount : balance;
```

We generate a boolean value.

The value to assign if the boolean value generated is true.

*'Assign our new balance to be the balance - amount, if the new balance would be greater than zero, otherwise simply reassign the same balance.'*



# ALTERNATIVE CONDITIONAL SYNTAX (2): SWITCH STATEMENTS

We also have the opportunity to test for the presence of a wide **range of values** with a **small amount of syntax** using a **switch statement**.

*Break statements give us the flexibility to have the same outcome for multiple cases.*

```
switch (pin) {  
    case 1234: balance = balance - amount;  
    break;  
    . . .  
    default: balance = balance;  
    break;  
}
```

*We also have the option to supply a **default** outcome, if none of the conditions are matched.*





# Iterative Statements

---

# REMEMBER: CODE REPETITION (1)

# Remember my self-centred code?

[illegible]

## REMEMBER: CODE REPETITION (2)

In order to understand what methods give us in **practice** (repeatable sections of code), we captured this code in a method.

```
public class MartinPrinter {  
  
    public void printMartin() {  
  
        System.out.println("+-----+");  
        System.out.println("|Martin|");  
        System.out.println("+-----+");  
  
    }  
  
}
```

*Remember that methods also provide us with a **separation of functionality**, which we have now seen more examples of (e.g. deposit and withdraw within a bank account).*

## REMEMBER: LABELLING CODE (1)

But remember we still had to **call** the method multiple times, which doesn't reduce the number of lines we need significantly:

*`Print Martin':*

```
System.out.println("+-----+");  
System.out.println("|Martin|");  
System.out.println("+-----+");
```

*Go to `Print Martin'*

*Go to `Print Martin'*

*Go to `Print Martin'*

## REMEMBER: LABELLING CODE (2)

It would be nice if we could do the following:

*`Print Martin':*

```
System.out.println("+-----+");  
System.out.println("|Martin|");  
System.out.println("+-----+");
```

*Do the following 3 times:* 

*Go to `Print Martin'*

*Remember: Just because the JVM is instructed to move to a different place in the program, this does not mean it can simply ignore what it was asked to do previously (in this case, to call Print Martin three times).*

We can achieve this by adding a **loop** into our code:

```
public class MartinPrinter {  
  
    public void printMartin() {  
  
        System.out.println("+-----+");  
        System.out.pr  
        System.out.pr  
  
    }  
  
}
```

```
public class Driver {  
  
    public static void main(String[] args) {  
  
        MartinPrinter martinPrinter = new MartinPrinter();  
        for ( int index = 0; index < 3; index = index + 1 ) {  
  
            martinPrinter.printMartin();  
  
        }  
  
    }  
  
}
```

# LOOP INDICES (1)

Loops allow us to **repeat** sections of our code.

However, unlike method calls, we can make this repetition happen **automatically**, as once we **start** a loop, we **can** ensure that it **runs by itself**, until it reaches a **stopping condition**.

To control this automatic execution, we **usually** declare a special **index** variable **within the loop itself**, which can be used by the loop to **track its own progress**.

```
for ( int index = 0; index < 3; index = index + 1 ) {  
    martinPrinter.printMartin();  
}
```

*In Computer Science,  
we often start counting  
from zero.*



## ASIDE: THE VARIOUS USES OF VARIABLES

We have now seen variables used in a number of settings:

- Storing **primitive** data.
- Storing copies of classes (as **objects**).
- As **parameters** in a method.
- Tracking the **index** of a loop.



# THE ANATOMY OF A FOR LOOP

A loop statement typically has **three** distinct sections to it.

Using the information in these sections, we can specify how many **times** our loop, and thus the code inside it, should **repeat**.

We declare our index variable to be a specific **value** (usually 0).

While this **boolean** condition (usually related to the index) is satisfied, the loop will continue running.

We use this section to instruct the loop on how to **change** the index variable, and thus **approach** its terminating condition.

```
for ( int index = 0; index < 3; index = index + 1 ) {
```

```
    martinPrinter.printMartin();
```

```
}
```

*The sections of our loop are separated by semi-colons.*

*Because we usually loop **for** a certain number of iterations, we call this a for loop.*

# LOOP INDICES (2)

**Index**

**Loop Iteration**

0

1

1

2

2

3

3

-

```
for ( int index = 0; index < 3; index = index + 1 ) {
```

```
    martinPrinter.printMartin();
```

```
}
```



## ASIDE: MIND YOUR BOUNDS

It's important to carefully check the upper and lower bounds when examining loop statements you find.

Slight changes to the the bounds of the loop can change the number of iterations...

```
for ( int index = 0; index <= 3; index = index + 1 ) {
```

...or not...

```
for ( int index = 1; index <= 3; index = index + 1 ) {
```

Draw tables for each of these loops in order to determine how many times they iterate, and the respective values of the index variable.

## ASIDE: PREFIX AND POSTFIX INCREMENT (1)

```
index = index + 1
```

It is so common to perform a single incrementation such as this, that Java provides us with a **shorthand** way to express the same operation in our programs:

```
index++;
```

These increments are actually another type of operator; a **unary** operator which acts on a single input.

## ASIDE: PREFIX AND POSTFIX INCREMENT (2)

```
index++;
```

```
++index;
```

We actually have two choices about where we supply the **`input`** to this operator: either before the operator (as we have seen) or after.

We call the former a **postfix** increment, and the latter a **prefix** increment.

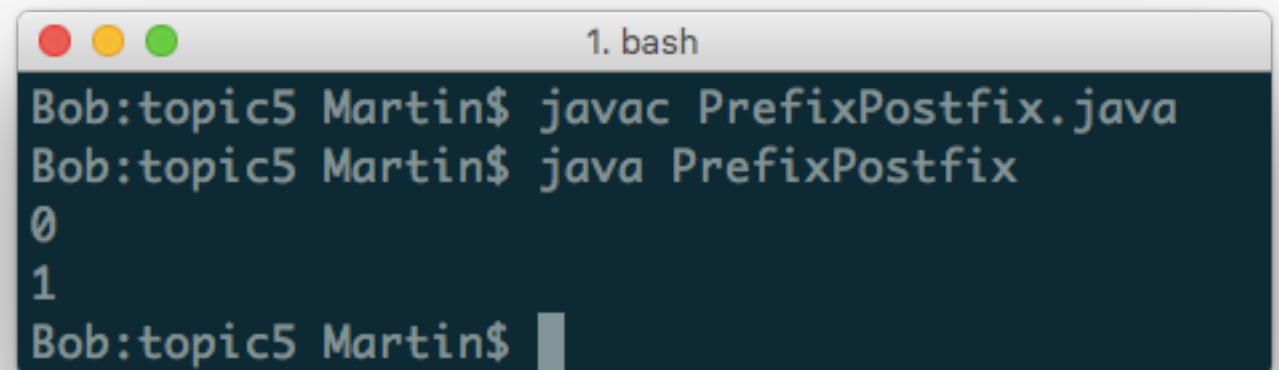
If we use a postfix increment, the value of the variable will only be incremented **after the statement in which the postfix increment appears has been evaluated**.

If we use a prefix increment, the value of the variable will be incremented **before the statement in which the prefix increment appears is evaluated**.

## ASIDE: PREFIX AND POSTFIX INCREMENT (3)

It's easy to confirm this using a simple print statement:

```
int x = 0, y = 0;  
System.out.println(x++);  
System.out.println(++y);
```

A terminal window titled "1. bash" showing the execution of the Java code. The prompt is "Bob:topic5 Martin\$". The first command is "javac PrefixPostfix.java". The second command is "java PrefixPostfix". The output is "0" followed by "1" on the next line. The prompt is "Bob:topic5 Martin\$".

```
1. bash  
Bob:topic5 Martin$ javac PrefixPostfix.java  
Bob:topic5 Martin$ java PrefixPostfix  
0  
1  
Bob:topic5 Martin$
```

However, in a for loop, whether we select a pre or a postfix increment is **unimportant** because we **don't use the value generated by the increment within the increment statement itself**.

```
for ( int index = 0; index <= 3; index++ ) {
```

## ASIDE: DECREMENTING LOOPS

As is probably clear, it's not always the case that the index values in our loops **increase**, we can, if we choose to, also instruct our loop to **decrease** an index value:

```
for ( int index = 3; index >= 0; index-- ) {
```



*We also have shorthand decrement.*

## ASIDE: ASSIGNMENT OPERATORS

In a similar vein, we have seen a more general form of incrementation in our bank account example, that adds a variable amount to an existing value:

```
balance = balance + amount;
```

Like the pre and postfix increments seen previously, we can also shorten this statement to:

```
balance += amount;
```

And we have a decrement equivalent:

```
balance -= amount;
```



Let's write a class to complement our `BookReader` called `AutomaticBookReader`, which contains a method called `readBook` that takes a book as a parameter, and reads a number of pages from that book, as also specified in a parameter.

Before each page is read, the program should print the page we are currently on.

```
public class AutomaticBookReader {  
    public void readBook(Book book, int pages) {  
        for ( int i = 0; i < pages; i++ ) {  
            System.out.println("On page: " + book.readPageNumber());  
            book.readPage();  
        }  
    }  
}
```

*We often shorten the word  
index to `i`.*

## BACK TO BOOKS (3)

```
public class BookReader {  
  
    public static void main(String[] args) {  
  
        Book aGameOfThrones = new Book(704);  
        AutomaticBookReader automaticBookReader = new  
                                                    AutomaticBookReader();  
        automaticBookReader.readBook(aGameOfThrones, 10);  
        System.out.println("(Reading manually) on page: " +  
                            aGameOfThrones.readPageNumber());  
  
    }  
  
}
```

*Remember the number of pages read (the **state**) of the book remains the same after we've `sent` the book to the read book method, and received it back (Topic 4, Slides 87 - 89).*

## REMEMBER: LABELLING CODE (2)

Can we also use a loop to reduce the number of lines in our Number Printer example?

*`Print number`:*

```
System.out.println("+-----+");  
System.out.println("| <NUM> |");  
System.out.println("+-----+");
```

*Go to `Print number` set **NUM = 1***

*Go to `Print number` set **NUM = 2***

*Go to `Print number` set **NUM = 3***

*We don't just want to repeat the call to Print Number, we also want to **change something** on each loop.*

While the index declared within a loop is important for tracking the **progress** of said loop, we can also **leverage** this index in order to **reduce unnecessary repetition** while **also** using it to make **incremental** changes inside the loop.

# LEVERAGING THE LOOP INDEX (2)

```
public class NumberPrinter {  
  
    public void printNumber(int num) {  
  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
        System.out.println("+-----+");  
  
    }  
  
}
```

```
public class Driver {  
  
    public static void main(String[] args) {  
  
        NumberPrinter numberPrinter = new NumberPrinter();  
  
        numberPrinter.printNumber(1);  
        numberPrinter.printNumber(2);  
        numberPrinter.printNumber(3);  
  
    }  
  
}
```

# LEVERAGING THE LOOP INDEX (3)

```
public class NumberPrinter {  
  
    public void printNumber(int num) {  
  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
        System.out.println("+-----+");  
  
    }  
  
}
```

```
public class Driver {  
  
    public static void main(String[] args) {  
  
        NumberPrinter numberPrinter = new NumberPrinter();  
  
        for ( int i = 1; i < 4; i++ ) {  
  
            numberPrinter.printNumber(i);  
  
        }  
  
    }  
  
}
```

*As the loop index is just a variable declaration, it can be referenced **inside** the loop.*

## ASIDE: LOOP SCOPE

Recall that it is typically the case that variables are only in scope if they are referenced within the same **block**, or within a nested **block**.

- Consider calling a field from within a method.

The same is true of loop (and indeed if) blocks: declared variables can only be referenced **within that block**, or within nested blocks.

```
for ( int i = 1; i < 4; i++ ) {  
    System.out.println(i);  
}  
  
System.out.println(i);
```

*This code will **not compile** due to the final line.*



# LEVERAGING THE LOOP INDEX (2): DIFFERENT INCREMENTS

```
public class NumberPrinter {  
  
    public void printNumber(int num) {  
  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
        System.out.println("+-----+");  
  
    }  
  
}
```

*We can actually write anything in the final section of the for loop, so long as it is an **expression statement** (not an assignment or another control flow statement).*

*As is probably clear, we can also increment the loop as we wish, and thus have further **control** over the index we have available for use within the loop itself.*

```
public class Driver {  
  
    public static void main(String[] args) {  
  
        NumberPrinter numberPrinter = new NumberPrinter();  
  
        for ( int i = 1; i < 4; i += 2 ) {  
  
            numberPrinter.printNumber(i);  
  
        }  
  
    }  
  
}
```

*What will this print?*

Given the for loop syntax we've seen so far, here are some simple problems to solve during your lab session:

- Sum the number from 0 to N (**Aside:** Do we need a loop for this at all?).
- Print the multiples of 3 up until N.
- Find the first N **Fibonacci numbers**.

# LIFE WITHOUT INDICES (1)

Despite everything a loop index give us, sometimes, when we want code to be repeated, it's not always clear that there's a definitive index **driving** this repetition.

For example, let's consider our old friend `System.currentTimeMillis()` and how it can be used in conjunction with a loop to create a toy\* **timer**.

*\*A timer you wouldn't actually use in practice, because there are much neater and safer ways to control the passage of time in Java.*

# LIFE WITHOUT INDICES (2)

```
public class ToyJavaTimer {
```

(This code can be **shortened**).

We store our start time, and then use it to determine an end time.

```
    public void oneSecondTimer() {
```

```
        long start = System.currentTimeMillis();
        long end = start + 1000;
```

So we could just omit them!

```
        for (; System.currentTimeMillis() < end;) {
```

Keep looping while the current time is less than the end time:

We want to **block** our program from running.

```
            System.out.println("Timing...");
```

From our pseudocode, it should be clear what our loop condition is.

```
        }
```

```
        System.out.println("End!");
```

```
for (      ?      ; System.currentTimeMillis() < end;      ?      ) {
```

Remember a familiar for loop...

But unless we initialise our end variable in the loop declaration, it's less clear what the initialisation and increment sections of a traditional for loop should be replaced with.



# WACKY FOR LOOPS

```
for (;System.currentTimeMillis() < end;) {
```

Like the example seen previously, the for loop syntax is actually fairly versatile.

```
for (;;) {  
    printMartin();  
}
```

So much so, that we can omit all of the contents of a for loop declaration, leaving **only** the semi-colons.

In the laboratory, find out what this loop does, and experiment with omitting other parts of the for loop definition while observing the effect this has.

# FOR LOOPS VS. WHILE LOOPS: READABILITY

```
for (;System.currentTimeMillis() < end;) {
```

While this is a legal piece of Java syntax, it isn't particularly **readable**.

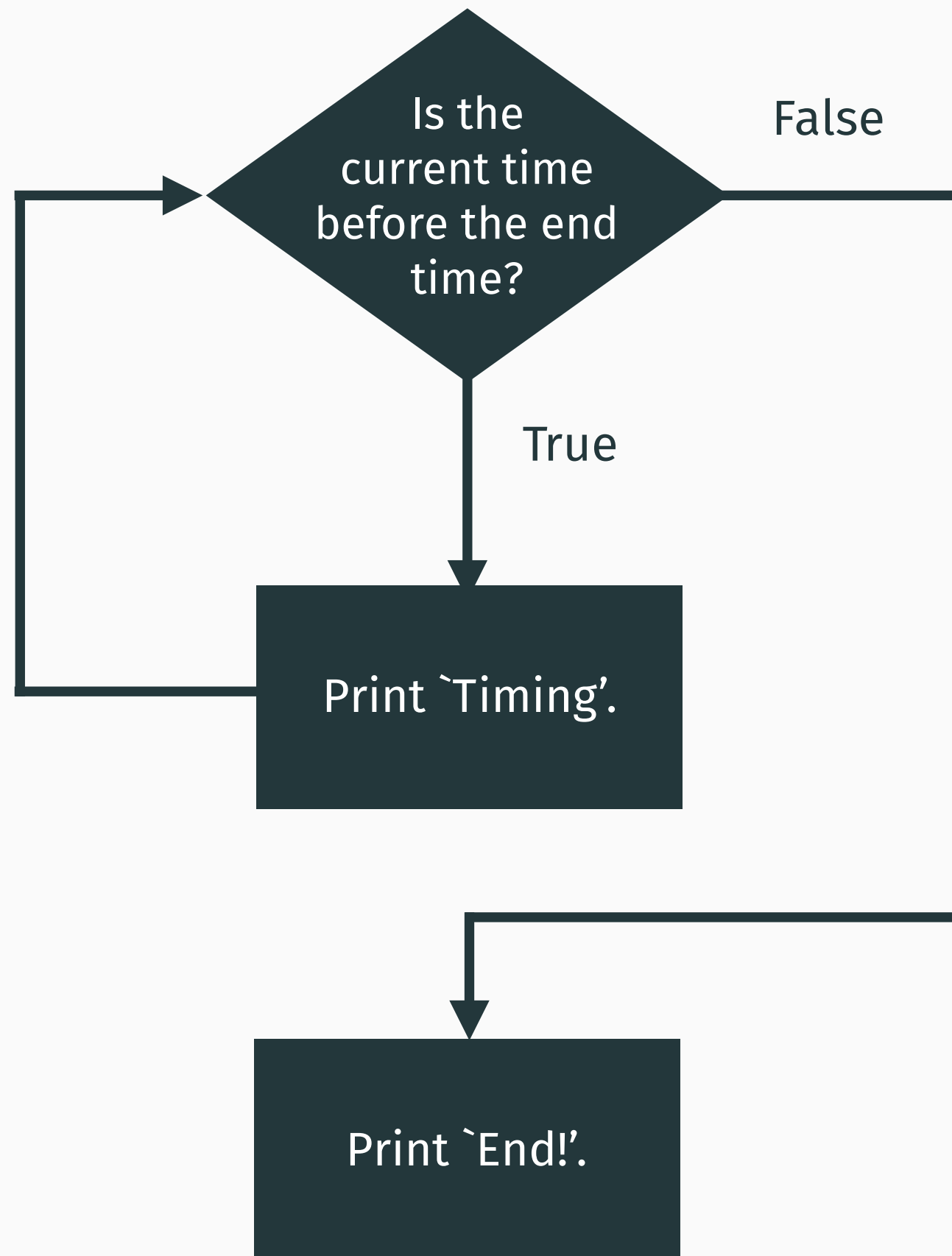
- It looks like the programmer has **missed something**.

For this reason, Java offers us another type of loop designed solely to cause code to repeat **while** a certain condition is **true**.

```
while (System.currentTimeMillis() < end) {
```

Rather than having three sections to its definition, this loop only retains one from the for loop: the boolean condition.

## ASIDE: THE BOOLEAN CONNECTION



# LIFE WITHOUT INDICES (3)

```
public class ToyJavaTimer {  
  
    public void oneSecondTimer() {  
  
        long start = System.currentTimeMillis();  
        long end = start + 1000;  
  
        while (System.currentTimeMillis() < end) {  
  
            System.out.println("Timing...");  
  
        }  
  
        System.out.pr  
  
    }  
  
}
```

```
public class Driver {  
  
    public static void main(String[] args) {  
  
        ToyJavaTimer toyJavaTimer = new ToyJavaTimer();  
  
        toyJavaTimer.oneSecondTimer();  
  
    }  
  
}
```



# ASIDE: LOOPS AND BRACES

In this example, we don't strictly need anything to output within our loop.

So, we could simply end our loop statement with a semi-colon, and omit the braces entirely.

```
public class ToyJavaTimer {  
  
    public class ToyJavaTimer {  
  
        public void oneSecondTimer() {  
  
            long start = System.currentTimeMillis();  
            long end = start + 1000;  
  
            while (System.currentTimeMillis() < end);  
  
            System.out.println("End!");  
  
        }  
  
    }  
  
}
```

*We **cannot** do the same thing with methods.*

*But be careful, I've seen lots of bugs arising from doing this unintentionally.*



# FOR LOOPS VS. WHILE LOOPS: SUMMARY (1)

For loops and while loops can be **semantically** equivalent.

```
for ( int index = 0; index < 3; index++) {  
    System.out.println(index);  
}
```

```
int index = 0;  
  
while ( index < 3 ) {  
    System.out.println(index);  
    index++;  
}
```

# FOR LOOPS VS. WHILE LOOPS: SUMMARY (2)

But for the purpose of **readability**, we typically employ them in specific **scenarios**:

- For loops, with an initialisation, a boolean condition, and an increment, when there are a **set** number of iterations we wish to perform.

```
for ( int index = 0; index < 3; index = index + 1 ) {
```

- While loops, with a single boolean condition, when we **do not know how many iterations** our loop will run for; we do not know at what point the boolean condition will **evaluate to false based on (complex) actions within the loop**.

```
while (System.currentTimeMillis() < end) {
```

*This is true of our timer with slight variations in the number of times the loop will repeat within the one second time span, depending on the decisions of the OS.*  99

# BACK TO BOOK READING (1)

For practice, let's rewrite the loop from our AutomaticBookReader as a while loop.

```
public class AutomaticBookReader {  
    public void readBook(Book book, int pages) {  
        for ( int i = 0; i < pages; i++ ) {  
            System.out.println("On page: " + book.readPageNumber());  
            book.readPage();  
        }  
    }  
}
```

## BACK TO BOOK READING (2)

```
public class AutomaticBookReader {  
  
    public void readBook(Book book, int pages) {  
  
        int i = 0;  
  
        while ( i < pages ) {  
  
            i++;  
            System.out.println("On page: " + book.readPageNumber());  
            book.readPage();  
  
        }  
  
    }  
  
}
```

# LOOPING WITHOUT AN INDEX

In the lab, try the following exercise, which more closely necessitate the use of a while loop.

- Determine how many times 5 can be subtracted from a number,  $N$ , before that number falls below zero.

And then restructure your previous for loops as while loops:

- Sum the numbers from 0 to  $N$ .
- Print the multiples of 3 up until  $N$ .
- Find the first  $N$  **Fibonacci numbers**.

# SPECIALIST LOOPS: DO WHILE AND THE ADVANCED FOR

There are two other loop types available to us, the **do while** loop and the **advanced for loop** but as these loops have **very specific** uses we will look at these in **Topic 6** and **Topic 7**, respectively.

There are two specific ways we can control the flow of execution within a loop:

```
continue;
```

```
break;
```

When we **continue**, we skip to the **next loop iteration**, updating indices as we do, if they exist.

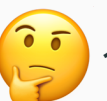
- We can use this to ignore sections of the code within a loop under various conditions.

When we **break**, we terminate our loop early.



Like the use of multiple returns within a method, both of these control structures cause **controversy** because they can result in **poorly designed code**.

- They work, but must be used **carefully**.
- As such, I won't discuss them in too much detail, but I want to draw your attention to their existence.



# Special Topic: Recursion

---

## ASIDE: DROPPING THE PRINT MARTIN METHOD

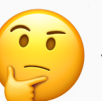
It may have seemed excessive, given our loop syntax, that we still have the `printMartin` method at all.

Instead, we could just call the code directly an arbitrary number of times using the appropriate loop:

*Do the following N times:*

```
System.out.println("+-----+");  
System.out.println("|Martin|");  
System.out.println("+-----+");
```

But then we would lose the ability to call `printMartin` from other places in our program.



# METHODS VS. LOOPS (1)

This opens up a wider debate about the use of methods and loops for repeating code.

Methods and loops both enable the same code to be used multiple times, and to be used in different ways, but they typically serve **different purposes**.

- Methods are called multiple times **manually** by referencing a label, whereas loops repeat **automatically** after writing the loop statement.
- Methods can be **referenced** at different times throughout a program, whereas the functionality within a loop is only referenced by the loop itself while it iterates (remember **scope**).

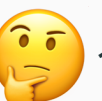


## METHODS VS. LOOPS (2)

This opens up a wider debate about the use of methods and loops for repeating code.

Methods and loops both enable the same code to be used multiple times, and to be used in different ways, but they typically serve different purposes.

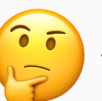
- Other code can happen between method calls, whereas the **iterations** of a loop typically happen in sequence.
- Methods are designed to capture sections of **functionality**, so also have a **structural** purpose.



# LOOPING WITH METHODS (1)

But it is true that loops and methods are closely connected concepts.

Because these are closely connected concepts, we can, in fact, **use methods to create a looping effect in our code.**



# LOOPING WITH METHODS (2)

```
for ( int i = 0; i <= 10; i ++ ) {  
    System.out.println(i);  
}
```

Let's consider a simple for loop.

How can we recreate **each element** of this loop using a method?



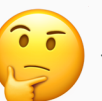
# LOOPING WITH METHODS (3): THE ITERATION

```
for (                                     ) {  
  
  
  
}
```

*In order to create the iterative effect of the loop, we have to **call the method within itself**.*

```
public void loop() {  
  
    loop();  
  
}
```

*If unchecked, this call process will naturally run on **indefinitely**.*





# LOOPING WITH METHODS (4): THE INDEX

```
for ( int i = 0;           ) {  
  
  
  
}
```

*In order to declare an index that will track the progress of our iterations, **we use a parameter.***

```
public void loopier(int i) {  
  
    loopier();  
}  
}
```

## LOOPING WITH METHODS (5): UPDATING THE INDEX

```
for ( int i = 0;           ; i ++ ) {  
  
  
  
}
```

Note the prefix increment here, otherwise our loop will run forever as the value of *i* will be lost in the **next call of the method**.

```
public void loopier(int i) {  
  
    loopier(++i);  
  
}  
}
```

*To update the index variable, we call the method with an incremented value on each iteration.*

# LOOPING WITH METHODS (6): THE LOOP CONDITION

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

*The loop condition is an if statement within the method, determining whether **another call to the method itself** will take place.*

```
public void looper(int i) {  
  
    if ( i <= 10 ) {  
  
        looper(++i);  
  
    }  
  
}
```

# LOOPING WITH METHODS (7): THE WORK

```
for ( int i = 0; i <= 10; i ++ ) {  
  
    System.out.println(i);  
  
}
```

```
public void loopier(int i) {  
  
    if ( i <= 10 ) {  
  
        System.out.println(i);  
        loopier(++i);  
  
    }  
  
}
```

*The lines from  
inside our loop  
typically appear  
**before** the  
method is  
called again.*

# RECURSION

Constructing a solution to a problem which involves a method calling itself is known as a **recursive** solution.

*In a recursive solution we typically have a **recursive case** which drives the iteration...*

```
public void loopier(int i) {  
    if ( i <= 10 ) {  
        System.out.println(i);  
        loopier(++i);  
    }  
}
```

*...and a **base case** which stops the iteration.*

Determining a suitable base and recursive case is an important task when constructing a recursive solution.

# STATEMENTS AFTER A RECURSIVE METHOD CALL

```
public void loopier(int i) {  
  
    if ( i <= 10 ) {  
  
        loopier(++i);  
        System.out.println(i);  
  
    }  
  
}
```

What happens if we do things **after** the recursive call?

- Why is this? **Hint:** remember, any statements we place in our code will be executed by the JVM, even if the flow of control is briefly redirected.
- Try this out in the laboratory, if you are interested.

Recursion can be used to create both **powerful** and **elegant** solutions to problems.

- You will see more of recursion next year in **4CCS1DST**.

# Topic 5: Control Flow

Programming Practice and Applications (4CCS1PPA)

---

Dr. Martin Chapman  
Thursday 20th October

[programming@kcl.ac.uk](mailto:programming@kcl.ac.uk)  
[martinchapman.co.uk/teaching](http://martinchapman.co.uk/teaching)

**These slides will be available on KEATS, but will be subject to ongoing amendments. Therefore, please always download a new version of these slides when approaching an assessed piece of work, or when preparing for a written assessment.**