Selection

Choices and some strings and characters

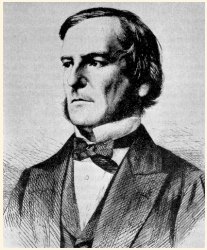
Tobias Andersson Gidlund

tobias.andersson.gidlund@lnu.se

Agenda

Boolean logic

- The foundation for both selection and iteration is *boolean logic*.
  - Invented by George Boole (1815 – 1865).
- Like all logic, it evaluates an expression to either *true* or *false*.
- What differs is that the used values in a boolean expression can only be 0 or 1 (false and true).
- After that, the connectives *not*, *and* as well as *or* can be used to put parts together.



Boolean expressions in Java

- In Java different variables with different values are evaluated.
  - The variables can have other values than 1 and 0 (true and false).
  - It is the evaluation of the expression that must become true or false.
- Three different types of operations are available to evaluate an expression:
  - Equality operations
  - Relational operators (size)
  - Logical operations (not, and as well as or)
- These can be put together in many different ways to be evaluated.
- Since many mathematical symbols are not available on the keyboard other combinations of characters are used.

Equality

- The equality tests if two values are equal.

```
number1 == number2;
```

- Notice that double equal signs are used – the single equal sign is used for assignment.
- The entire expression is evaluated to true when the value in `number1` is the same as `number2`.
- For not equal in Java the sign `!=` is used.
  - The mathematical symbol is $\neq$, that is not available on the keyboard.

```
number1 != number2;
```

- In this case the expression is *true* if the variables contain different values.

Relational operators

- It is also possible to compare *relations* between two values.
- The following operators are allowed:

Mathematics	Java	Meaning
$<$	$<$	Less than
$\leq$	$<=$	Less than or equal to
$>$	$>$	Greater than
$\geq$	$>=$	Greater than or equal to

- The expression:

```
aNumber <= anotherNumber
```

- Is true if `aNumber` is less than or equal to `anotherNumber`.
  - For example if `aNumber` is 3 and `anotherNumber` is 4.
  - Or if both variables have the same value.

Logical operators

- A part from the earlier operators, there are also *logical* operators.
- They are part of the boolean logic and they are the *connective* between parts.
  - They put two part together with a logical meaning.
- It is important to remember that the parts that are put together must be evaluated to *true* or *false*.
  - It is not, for example, possible to use integers as with relations and equality.
- After that the whole is evaluated.

The logical operators

- The logical operators are:

Logic	Java	Meaning
$\neg$	!	Not, negation
$\wedge$	&&	And
$\vee$	||	Or

- Please notice that the operators & and | also exist but that they mean something differently.
  - They are so called *bitwise* logical operators.
- The logical operators can be used together with equality and relation.

Meaning

- The meaning of the different connectives can be shown in a *truth table*.
- The truth tables for them are shown on the following slides in a way that all possible alternatives are shown.
- For negation, the table looks like this:

A	$\neg A$
s	f
f	s

- The table is read as "When A is true, then $\neg A$ is false".
- Since negation is *unary* (that is, only for one part) these two alternatives are the only possible.

Conjunction

- A conjunction is true only when both statements are true.
- The following is the truth table:

A	B	$A \wedge B$
s	s	s
s	f	f
f	s	f
f	f	f

Disjunction

- A disjunktion is true if at least one of A and B are true.
- Disjunction is also binary (two statements are needed) and therefore four lines.

A	B	$A \vee B$
s	s	s
s	f	s
f	s	s
f	f	f

Putting logical expressions together

- The different parts can then be put together to create longer logical expressions.
- Example:

```
aNumber > anotherNumber || anotherNumber > aNumber
```

- If `aNumber` is 3 and `anotherNumber` is 4, then the expression is *true*.
- But, how does Java know in what order to evaluate the parts?
- Java follow the mathematical *priority order*.
- This with an addition for the programming specific parts.

Priority order

- Operators have the following priority:
  - Parenthesis
  - Increase and decrease operators (++ and –)
  - Multiplication, division and modulus
  - Addition and subtraction
  - Relational operators
  - Equality operators
  - Logical operators
  - Assignment
- This makes it possible to chain several operators for longer calculations.

Selection

Selection

- Selection is about making a *choice* between different alternatives.
- This is something everyone is doing every day.
  - What is the right clothes for today?
  - Do I have enough money to buy a movie?
  - Should I talk the lift or the stairs?
  - Should I eat at home or out?
- Sometimes the choices are done explicitly, sometimes they are done by “habit”.
  - Picking of clothes can be either by choice or by picking the top in the drawer.
- Sometimes we know what is behind our choices, sometimes it is done sub-consciously.
  - This is called tacit knowledge.

Level and detail in selection

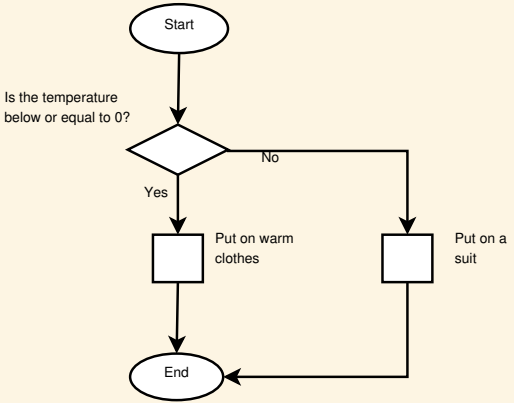
- The level and the amount of detail to put in a selection depends on the receiver.
- If the question is “What should I wear?”, then different answers are needed for:
  - The middle age man – it can suffice with “It is -7 degrees outside”.
  - The six year old – then the degrees are not sufficient, closer instructions are needed.
- For a choice to be carried out, it must be *unambiguous*.
  - It should not be possible to interpret the questions in any other way than intended.
- Different receivers need information on different levels.

Selection in code

- To create a selection in program code is to make it unambiguous.
- The first part of that is to make the condition to a boolean expression.
- This means that the question needs to be answerable using *true* or *false*.
- For the question about clothes, the first question can be: `Is the temperature below zero?`
- For this quesition, three answers are possible:
  - The temperature is below zero.
  - The temperature is above zero.
  - The temperature is is exactly zero.
- To create *two* answers for the condition the needs to be more precise. `Is the temperature zero or less?`

Flow chart

- The previous choice can be visualised as the following flow chart.



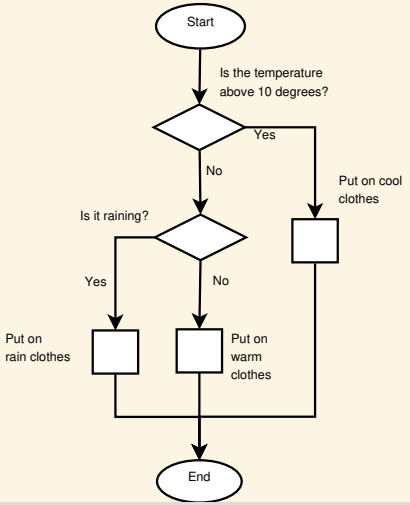
Questions in many steps

- It is possible to put many questions after each other to answer more questions.
- We need not answer all our questions, that is – we need not do anything for every true/false.

- The problem is that if the conditions do not change (that is, the temperature is rising), both choices will be performed.
  - This would mean that we need to put on *both* warm *and* really warm clothes.
- It is therefore possible to connect a number of questions to one and the same base condition, to have one question with *several* answers.

Example, nested questions

- In the example to the right there is a question in the question.
- In a flow chart it is easy to see that the question is only applicable if the temperature is below ten degrees.
  - This is called a *nested* question.
- To put on a rain coat or not is mutually exclusive.



Selection in Java

- In Java there are two statements that handle selection:
  - `if` statements
  - `switch` statements
- We will look further on the `if` statement as it most closely resembles the flows we have seen before.
- Just as for the flows, the condition for the selection must be constructed in a way that can be answered with either *true* or *false*.
- The condition is constructed using a *boolean expression* as has been discussed previously.
- If it is true then the statement or block closest to it will be executed.

`if` statements

- The structure of the `if` statement is:

```
if (boolean expression)
  One statement
```

- Note the indentation, a “tab step”. It is not necessary for Java, but makes the understanding of the statement easier.
- In this case only *one* statement can be performed after the choice.
- An example:

```
if(temp < 0)
  System.out.println("Lower than zero");
```

- The print out is only done if the variable `temp` has a value below zero.

More `if` statements

- It is possible to put several `if` statements after each others.
- All that are true will be performed.

```
package lecture3;
public class Temperature2 {
   public static void main(String[] args) {
  int temp = 21;

  if(temp > 10)
      System.out.println("Above 10 degrees, wear trousers.");
  if(temp > 20)
      System.out.println("Above 20 degrees, wear shorts");
  }
}
```

- Both print outs will be performed, which means that both trousers and shorts must be worn...

Else

- Since we want all the alternative to be mutually exclusive, we need to but them together.
  - As we did in the flow chart.
- In Java this is done using `else` followed by the statement that should be performed.

```
package lecture3;

public class Temperature3 {

  public static void main(String[] args) {
    int temp = 28;

    if(temp > 25)
      System.out.println("Wear shorts");
    else
      System.out.println("Wear trousers");
  }
}
```

More alternatives

- In addition to the single `else`, it is possible to create a longer question list.
- For every new condition a new `if` statement is needed.
  - Except the last, it will catch all other alternatives.
- The structure is as follows:

```
if (expression)
  statement
else if (expression)
  statement
else if (expression)
  statement
...
else
  statement
```

Example

```
package lecture3;
public class Temperature4 {
  public static void main(String[] args) {
    int temp = 3;

    if (temp < 0)
      System.out.println("Below zero");
    else if (temp == 0)
      System.out.println("Zero");
    else if (temp > 0)
      System.out.println("Above zero");
    else
      System.out.println("Not a chance...");

  }
}
```

- Output:

  ```
  Above zero
  ```

Only one alternative will be performed

- In the example all the alternatives were mutually exclusive.
- But, what happens if they are not?
- If we put the following `else if` after `else if (temp > 0)` and change the temperature to 30.

```
else if (temp > 28)
  System.out.println("Really warm");
```

- Will then the output be `Really warm`?
- No – only the first alternative to meet the condition will be performed.
- Presently it is not possible to test both above zero and above 28.

Boolean expressions, again

- As previously stated, the condition is a boolean expression.
- With the help of those, we can state a condition that consists of more than one expression.
- To pose the right question, the condition “above zero” needs to be put in the range 0 to 27.
  - In mathematical terms as $0 < temp > 27$.
- In Java it is not possible to use the mathematical way, so we need to rewrite it using the boolean connective “and”.

```
else if (temp > 0 && temp < 27)
```

- At the same time we change the second selection to be for 28 or above or just 28.

```
else if (temp >= 28)
```

Example

```
package lecture3;

public class Temperature5 {
  public static void main(String[] args) {
    int temp = 30;

    if (temp < 0)
      System.out.println("Below zero");
    else if (temp == 0)
      System.out.println("Zero");
    else if (temp > 0 && temp < 27)
      System.out.println("Above zero");
    else if (temp >= 28)
      System.out.println("Really warm");
    else
      System.out.println("Not a chance...");
  }
}
```

- Output: `Really warm`

More statements for each alternative

- So far only one statement has been performed for each alternative.
- It is possible, though, to have several statements put in a *block*.
- A block begins with a start curly bracket and ends with a ending curly bracket.
  - Just as a class or a method.
- The first curly bracket can be put either on the same line as the `if` statement or on the next.
  - In the beginning the practice was to put it on the same line, but more and more are now using the other form (originating from C).
- Inside of the curly brackets as many statements as possible can be put.

```java
package lecture3;

public class Temperature6 {
  public static void main(String[] args) {
    int temp = -10;
    boolean snow = false;

    if(temp < -20)
    {
      System.out.println("Quite cold");
      snow = true;
    }
    else if (temp <= 0)
    {
      System.out.println("Cold");
      snow = true;
    }
    else if (temp >= 0 && temp <28)
    {
      System.out.println("Warm");
    }
    else
      System.out.println("Really warm");

    if(snow)
      System.out.println("Chance of snow");
    else
      System.out.println("No chance of snow");
  }
}
```

```java
package lecture3;

public class Temperature6 {
  public static void main(String[] args) {
    int temp = -10;
    boolean snow = false;

    if(temp < -20) {
      System.out.println("Quite cold");
      snow = true;
    }
    else if (temp <= 0) {
      System.out.println("Cold");
      snow = true;
    }
    else if (temp >= 0 && temp <28) {
      System.out.println("Warm");
    }
    else
      System.out.println("Really warm");

    if(snow)
      System.out.println("Chance of snow");
    else
      System.out.println("No chance of snow");
  }
}
```

Nested `if` statements

- A `if` statement can also be inside of another `if` statement.
  - It is then a nested `if` statement.
- This means that the inner `if` statement only can be performed if the outer is true.
- A program can have as many levels of nesting as you like, but already after a few it will become hard to keep track of them.
- To make it easier to see what statement a specific `if` belongs to, it recommended to always use blocks for nested `if` statements.
  - It is also necessary for Java in order to be certain what `if` a specific `else` belongs to.
  - A `else` is always set to the last seen `if`, but sometimes this is not what we mean.

Example

```
package lecture3;

public class Temperature7 {
  public static void main(String[] args) {
    int temp = 10;
    boolean rain = true;
    boolean umbrella = false;

    if(temp < 0)
      System.out.println("Cold, but not raining");
    else if (temp > 0 && temp < 20) {
      if(!rain)
        System.out.println("Clear sky");
      else if (rain && umbrella)
        System.out.println("It is raining, use your umbrella");
      else if (rain && !umbrella)
        System.out.println("it is raining and you will get wet");
    }
    else
      System.out.println("Hot and dry");
  }
}
```

Switch Statement

- The switch statement chooses case based on a value

```
switch (value) {
   case value1:        // If value1 = value
      do something
      break;           // jumps to next statement
   case value2:
      do something
      break;
   default:
      other values are handled here
}
next statement;
```

- Possible value types: integers (`long`, `int`, `short`, `byte`) and characters (`char`) and also strings.
- Chooses the first `case` if there are several matches
- Skip `break` → we continue to next `case` → complicated code
- Skip `default` → like an `if` statement without an `else`
- Between `case` and `break` there could be several statements

Example: switch statement

```
Scanner scan = new Scanner(System.in);
System.out.print("Give a weekday number: ");
int dayNumber = scan.nextInt();

String weekDay;
switch (dayNumber) {
  case 1:
     weekDay = "Monday";
     break;
  case 2:
     weekDay = "Tuesday";
     break;          // Skipping cases 3-6
  case 7:
     weekDay = "Sunday";
     break;
  default:
     weekDay = "Incorrect weekday number";
}
System.out.println("Weekday: " + weekDay);
```

The Conditional Operator ?:

The conditional operator `?:` is similar to the `if-else` statement.
Difference: it will return a value.

```
int max = (int1 > int2) ? int1 : int2;
```

Above: `max` is assigned the largest of the values `int1` and `int2`.

- Generally for the operator `?:`

```
boolExpr ? Expr1 : Expr2;
```

- `boolExpr` is `true` → `Expr1` is computed and returned
- `boolExpr` is `false` → `Expr2` is computed and returned

Printing example:

```
System.out.println("The integer "+N+" is "+( (N%2==0)?"even":"odd" ) );
```

The conditional operator `?:` should be used with consideration. It might give rise to unreadable code. Use `if-else` if the involved expressions are complicated.

Mathematical Functions

Common mathematical functions

- Something often done in computer programs is to perform some sort of *calculation*.
  - The first computers were only advanced calculators.
- To make it easier to make calculations, Java has a number of predefined functions to use.
- A number of *static* members are available in the `Math` class, for example an approximation of π is given by `Math.PI`:

```
System.out.println("Pi is approximately " + Math.PI);
```

- Which gives the output:

```
Pi is approximately 3.141592653589793
```

Other functions

- The book lists almost all of the functions available in Java.
- An example: the area of a circle is calulated as $Area = \pi r^2$
- In Java it will become:

```
double radius = 4.0;

double area = Math.PI * Math.pow(radius, 2.0);

System.out.println("A circle with the radius " + radius + " has an area of " + area);
```

- The output is:

```
A circle with the radius 4.0 has an area of 50.26548245743669
```

- Notice how the function `Math.pow()` takes two parameters, one for the base and one for the exponent.

More methods of `Math`

- There are also four useful methods for rounding numbers.
  - `ceil(x)` for rounding up to the nearest integer but as a double value.
  - `floor(x)` for rounding down to the nearest integer but as a double value.
  - `rint(x)` to round to the nearest integer but as a double value.
  - `round(x)` round to the nearest integer but as the type `x` is.
- This can be useful also for rounding to different decimal places as the example will show.

Example

```
public class TestingRounding {
    public static void main(String[] args) {
        System.out.println(Math.ceil(2.345));
        System.out.println(Math.floor(2.345));
        System.out.println(Math.rint(2.345));
        System.out.println(Math.round(2.345));
        System.out.println(Math.round(2.365 * 10.0) / 10.0);
    }
}
```

Printout:

```
3.0
2.0
2.0
2
2.4
```

Random

- The class `Random` is used to generate random numbers.
- There are many situations where random numbers are interesting:
  - Enemy movement in a game
  - Generate lotto lines
  - Create difficult do break cryptos
- The class is available in `java.util.Random` and needs to be imported before used.
- This class is a *pseudo random generator* – it does not give "real" random numbers.
  - Uses a *random seed* – often based on time – and picks a number based on some calculations.
- It is, however, random enough for most applications.

Using random numbers

- An object must be created of the class using new.

```
Random rnd = new Random();
```

- After that a number of methods can be used:
  - `double nextDouble()` – returns a number between 0.0 (inclusive) and 1.0 (exclusive).
  - `int nextInt(int n)` – returns an integer between 0 (inclusive) and *n* (exclusive).
- Example:

```
int aNumber = rnd.nextInt(10);
```

- Gives a random number between 0 and 9 which is put in `aNumber`,

Example

```
package lecture3;
import java.util.Random;

public class RandomCharacters {
  public static void main(String[] args) {
    Random rnd = new Random();

    double height;
    int age;

    age = rnd.nextInt(100);
    height = 100 + rnd.nextDouble() * 100.0;

    System.out.print("Your character is " +age+ " years old");
    System.out.println(" and " + height + " meters tall.");
  }
}
```

Characters and Strings

Characters

- `String` has previously been used, but it is not a primitive type, but it is built by the primitive type **`char`**.
- This data type is used to represent a single character.
- In contrast to `String` single quotes are used to represent a character.
- The data type is 16 bit large to be able to contain 65536 different characters.
  - Each printable (and some non-printable) character is represented by a number between 0 and 65535.
- This means that many different characters can be printed, for example the Swedish åääö.
  - **`char`** follows the *Unicode character set*.

Example

```
package lecture3;

public class Characters {
  public static void main(String[] args) {
    char letterA = 'a';
    char newLine = '\n';
    char hengHook = '\u0267';

    System.out.print(letterA+" string "+newLine+hengHook);
  }
}
```

Output

```
a string
ɧ
```

Escape sequences

- Since quotation marks define a string, it makes it impossible to write a quotation mark using `"""`.
- But, to be able to write such characters, Java uses *escape sequences*.
- These are instructions with special meaning.
  - Called “escape” since they escape the normal flow in a string.
- To print a quotation mark, use the *character* `\"`.
  - This is seen as *one* character.
- All escape sequences begin with a backslash (`\`) followed by another character.
  - When the string is printed the two signs are converted into one single character.

Escape sequences

- These are the escape sequences used in Java:

Sequence	and meaning
\b	back step
\t	tabulator
\n	new line
\r	return
\”	double quotation
\’	single quotation
\\	backslash

- They can be put anywhere in a string.

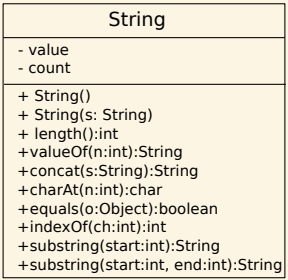
Example

```
package lecture3;

public class Albums {
  public static void main(String[] args) {
    System.out.println("Really good music albums\n");
    System.out.println("Artist\t\tAlbum");
    System.out.println("Pink Floyd\t\"Wish You Were Here\"");
    System.out.println("Pink Floyd\t\"The Wall\"");
    System.out.println("And One\t\t\"S.T.O.P.");
    System.out.println("Pakt\t\t\"Berlin\"");
  }
}
```

```
Really good music albums

Artist          Album
Pink Floyd      "Wish You Were Here"
Pink Floyd      "Dark Side of the Moon"
And One         "S.T.O.P.
Pakt            "Berlin"
```

Strings

- Strings of the class `String` are real objects.
- They do, however, have several properties that make them look like primitive types.
  - Assignment is, for example, possible:
    `String str = "Star Wars";`
- Strings are important as they are the first *data structure* that we use.
  - A data structure is a description of how some data is organised.
- A string is a number of `characters` organised in a `sequence`.



Simplified image of `String` (more methods exist).

Immutable

- Strings are immutable, once they have been created they cannot be changed.
  - Which is the meaning of *immutable*, not able to be changed.
- Several of the methods on the previous slide did give the impression that they could.
  - `concat` adds one string to another.
  - The book shows several more, like `replace` and `toLowerCase`.
- Common with these is that they *construct* a new string which is returned.
- This means that if such a method is used, a new object is created in memory and a reference to the new memory is returned.
  - Often the old value is overwritten, but that is not mandatory.

```java
public class StringConcat {
  public static void main(String[] args) {
    String part1 = new String("Pink");
    String space = new String(" ");
    String part2 = "Floyd";
    String pf = new String();

    // Create a new string from part1 and space
    pf = part1.concat(space);

    // Create a new string from pf and part2
    pf = pf.concat(part2);

    System.out.println("The best band: " + pf);

    // part1, part2 and space are not changed
    System.out.println(part1 + space + part2);
  }
}
```

```
The best band: Pink Floyd
Pink Floyd
```

More methods

- String concatenation can be done using the `concat` method or using the plus operation.
  - `pf = part1.concat(space);` and `pf = part1 + space;` gives the same result.
- Which is “best”? It depends...
- The book disusses more methods and how they work.
  - The methods either return a value or creates a new and changed object.
- As strings are important (they are used a lot), it is important to learn how they work.

Example

```
package lecture3;

public class InAWorldOfStrings {
  public static void main(String[] args) {
    String author = new String("Isaac Asimov");
    String book = new String("Foundation ");

    String bigBook = book.toUpperCase();
    String tabbedBook = bigBook.replace(' ', '\t');

    System.out.println("Book: " + tabbedBook + author);
    System.out.println("File under: " +
      author.substring(6, author.length()));
  }
}
```

```
Book: FOUNDATION    Isaac Asimov
File under: Asimov
```

A sequence of characters

- The concept of *arrays* is going to be addressed later in the course.
  - It is a data type which holds a sequence of other data types reachable via index numbers.
- As stated earlier, a string is a sequence of characters.
- It is possible to reach each character in a string.
- Each character is given an *index number*.

- Notice that zero is the first index number.
- The method `charAt(x)` can be used to retrieve the character at index position `x`.

Example

```
package lecture3;

public class InTheString {
  public static void main(String[] args) {
    String alpha = new String("ABCDEFGHIJKLMNOPQRSTUVWXYZ");

    System.out.println("First letter: " + alpha.charAt(0));
    System.out.println("Last letter: "
        + alpha.charAt(alpha.length()-1));

    System.out.print("The master: " + alpha.charAt(24) +
      alpha.charAt(14) + alpha.charAt(3) + alpha.charAt(0));
  }
}
```

```
First letter: A
Last letter: Z
The master: YODA
```

Summary

- This lecture has touched on two very important concepts:
  - Selection
  - Strings (and characters)
- Both concepts will follow the entire course (and also in the next).
- The book looks at a number of additional methods and properties for both, read and try them out.
- There is nothing to make you understand as much as doing, so use your IDE and do the examples in the book!