

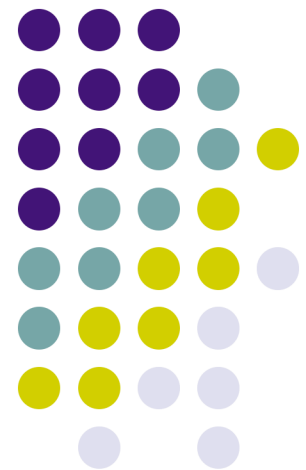


Principles of Computer Science I

Prof. Nadeem Abdul Hamid

CSC 120 – Fall 2006

Lecture Unit 6 - Decisions





Lecture Outline

- Implementing decisions using if statements
- Grouping statements into blocks
- Comparing numbers, strings, and objects
- Using Boolean operators and variables

Making Decisions



Making Decisions



- Computer programs often need to make decisions
 - Take different actions depending on some condition(s)



Making Decisions

- Computer programs often need to make decisions
 - Take different actions depending on some condition(s)
- Example: Can't withdraw more money than in account balance
 - “If amount-to-withdraw is less than available balance then deduct from balance; otherwise charge a penalty to the balance.”



Making Decisions

- Computer programs often need to make decisions
 - Take different actions depending on some condition(s)
- Example: Can't withdraw more money than in account balance
 - “If amount-to-withdraw is less than available balance then deduct from balance; otherwise charge a penalty to the balance.”
- ```
if (amount <= balance)
 balance = balance - amount;
```



# if/else Statement

- Does this work?

```
if (amount <= balance)
 balance = balance - amount;
if (amount > balance)
 balance = balance - OVERDRAFT_PENALTY;
```

- How about this?

```
if (amount <= balance)
 balance = balance - amount;
else
 balance = balance - OVERDRAFT_PENALTY;
```



# Types of Statements

- Simple
  - `balance = balance - amount;`
- Compound
  - `if ( amount <= balance ) balance = balance - amount;`
- Block
  - Groups multiple statements together
  - Can be used anywhere a single statement is used

```
{
 double newBalance = balance - amount;
 balance = newBalance;
}
```





# Syntax: if Statement

```
if (condition) statement
```

```
if (condition) statement1 else statement2
```

## **Purpose:**

To execute a statement(s) depending on whether a condition is true or false



# Syntax: Block Statement

```
{
 statement1
 statement2
 ...
}
```

## **Purpose:**

To group several statements together to form a single statement



# Brace Layout

- Doesn't matter to compiler – matters to human
- Two suggested styles – choose one and stick to it

```
if (amount <= balance)
{
 double newBalance = balance - amount;
 balance = newBalance;
}
```

- **or**

```
if (amount <= balance) {
 double newBalance = balance - amount;
 balance = newBalance;
}
```



# Indentation

- Another very critical way to make programs readable for humans
- Use spaces instead of tab key
- 2, 3, or 4 spaces are best
- Tips
  - Always type the beginning and ending braces first, then fill in between
  - Put comment after closing brace to indicate what it matches



# Indentation

- Another very critical way to make programs readable for humans
- Use spaces instead of tab key
- 2, 3, or 4 spaces are best
- Tips
  - Always type the beginning and ending braces first, then fill in between
  - Put comment after closing brace to indicate what it matches

```
public class BankAccount {
 . . .
 public void withdraw(double amt)
 {
 if (amt <= balance)
 {
 double newBal = balance - amt;
 balance = newBal;
 }
 . . .
 }
}
```



# Indentation

- Another very critical way to make programs readable for humans
- Use spaces instead of tab key
- 2, 3, or 4 spaces are best
- Tips
  - Always type the beginning and ending braces first, then fill in between
  - Put comment after closing brace to indicate what it matches

```
public class BankAccount {
 . . .
 public void withdraw(double amt)
 {
 if (amt <= balance)
 {
 double newBal = balance - amt;
 balance = newBal;
 }
 . . .
 }
}
```

```
public class BankAccount
{
 . . .
 public void withdraw(double amt)
 {
 if (amt <= balance)
 {
 double newBal = balance - amt;
 balance = newBal;
 } // end if
 } // end withdraw method
 . . .
} // end BankAccount class
```



# Comparing Values

- *Relational* operators

| <i>Java</i> | <i>Math Notation</i> | <i>Description</i>    |
|-------------|----------------------|-----------------------|
| >           | >                    | Greater than          |
| >=          | ≥                    | Greater than or equal |
| <           | <                    | Less than             |
| <=          | ≤                    | Less than or equal    |
| ==          | =                    | Equal                 |
| !=          | ≠                    | Not equal             |

- == operator denotes equality testing

```
a = 5; // Assign 5 to a
if (a == 5) . . . // Test whether a equals 5
```

# Comparing Floating Point



```
double r = Math.sqrt(2);
double d = r * r - 2;
if (d == 0)
 System.out.println("sqrt(2)squared minus 2 is 0");
else
 System.out.println("sqrt(2)squared minus 2 is not 0 but " + d);
```



# Comparing Floating Point



```
double r = Math.sqrt(2);
double d = r * r - 2;
if (d == 0)
 System.out.println("sqrt(2)squared minus 2 is 0");
else
 System.out.println("sqrt(2)squared minus 2 is not 0 but " + d);
```

sqrt(2)squared minus 2 is not 0 but 4.440892098500626E-16



# Comparing Floating Point

```
double r = Math.sqrt(2);
double d = r * r - 2;
if (d == 0)
 System.out.println("sqrt(2)squared minus 2 is 0");
else
 System.out.println("sqrt(2)squared minus 2 is not 0 but " + d);
```

sqrt(2)squared minus 2 is not 0 but 4.440892098500626E-16

- Don't compare floating point numbers for (exact) equality ==
  - Doesn't work because of roundoff errors
- Instead, check if they are close enough (up to a desired threshold)

# Comparing Floating Point (Correctly)



- Test whether (absolute value of) the difference between two number is close to 0
  - Threshold often referred to as  $\varepsilon$  – ‘epsilon’

$$|x - y| \leq \varepsilon$$

- In Java:

```
final double EPSILON = 1E-14;
if (Math.abs(x - y) <= EPSILON)
 // x is approximately equal to y
```



# Comparing Strings

- Don't use `==` for strings either!

```
if (input == "Y") // WRONG!!!
```

- Use the `equals` method

```
if (input.equals("Y")) . . .
```

- `==` tests identity; `equals` tests equal contents
  - Will see this again in 'Comparing Objects' slides

- To test equality ignoring upper/lowercase ('Y' or 'y')

```
if (input.equalsIgnoreCase("Y")) . . .
```



# Comparing Order of Strings

- Use the `compareTo` method
  - `s.compareTo(t) < 0` means s comes before t
  - `s.compareTo(t) > 0` means s comes after t
  - `s.compareTo(t) == 0` means s and t are equal
- Java's 'dictionary' order is according to Unicode
  - 'car' comes before 'cargo'
  - All uppercase letters come before lowercase
    - 'Hello' comes before 'car'
  - Numbers come before letters
    - '1' comes before 'a'
- See Appendix B in textbook



# Comparing Objects

- Like strings, `==` tests identity; `equals` tests contents

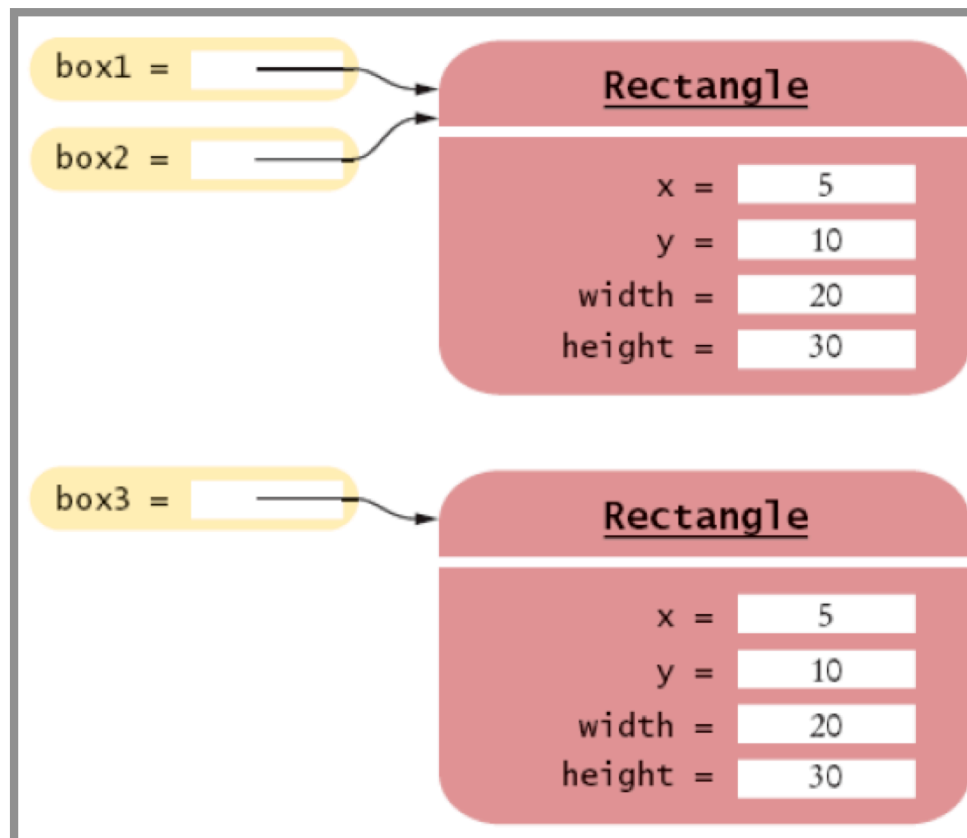
```
Rectangle box1 = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box1;
Rectangle box3 = new Rectangle(5, 10, 20, 30);
```

- `box1 != box3` but `box1.equals( box3 )`
- `box1 == box2`
- Warning: `equals` method must be defined properly by the class before you can use it



# Object References

```
Rectangle box1 = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box1;
Rectangle box3 = new Rectangle(5, 10, 20, 30);
```





# Testing for null

- Object variable may be set to `null`
- Indicates 'no object'

```
String middleInitial = null; // Not set
if (. . .)
 middleInitial = middleName.substring(0, 1);
```

- Can be used as a condition (use `==`):

```
if (middleInitial == null)
 System.out.println(firstName + " " + lastName);
else
 System.out.println(firstName + " " + middleInitial
 + ". " + lastName);
```





# Strings and null

- Empty string is ""
  - Valid string of length 0
- `null` indicates a string variable does not refer to anything, not even an empty string
- Always test for `null` using `==` not the `equals` method



# Conditions with Side Effects

- Avoid in `if` statements!
  - Bad programming practice
- Side effects: assignment, increment, decrement

```
if ((d = b * b - 4 * a * c) >= 0) r = Math.sqrt(d);
```

```
if (n-- > 0) . . .
```

- Can occasionally be useful to simplify loops
  - Next chapter



# Multiple Alternatives

- Sequences of comparisons

```
if (condition1) statement1;
else if (condition2) statement2;
.
.
.
else statementN;
```

[Earthquake.java](#)

[EarthquakeTester.java](#)

- The first matching condition is executed
- Order matters!

```
if (richter >= 0) // always passes
 r = "Generally not felt by people";
else if (richter >= 3.5) // not tested
 r = "Felt by many people, no destruction. . ."
```



# if vs. if/else

- Consider carefully which one is appropriate to use

```
if (richter >= 8.0)
 r = "Most structures fall";
if (richter >= 7.0)
 r = "Many buildings destroyed";
if (richter >= 6.0)
 r = "Many buildings considerably damaged, some collapse";
if (richter >= 4.5)
 r = "Damage to poorly constructed buildings";
if (richter >= 3.5)
 r = "Felt by many people, no destruction";
if (richter >= 0)
 r = "Generally not felt by people";
return r;
```



# Nested Branches

- One `if` statement inside another

```
if (condition1) {
 if (condition1A)
 statement1A;
 else
 statement1B;
} else
 statement2;
```

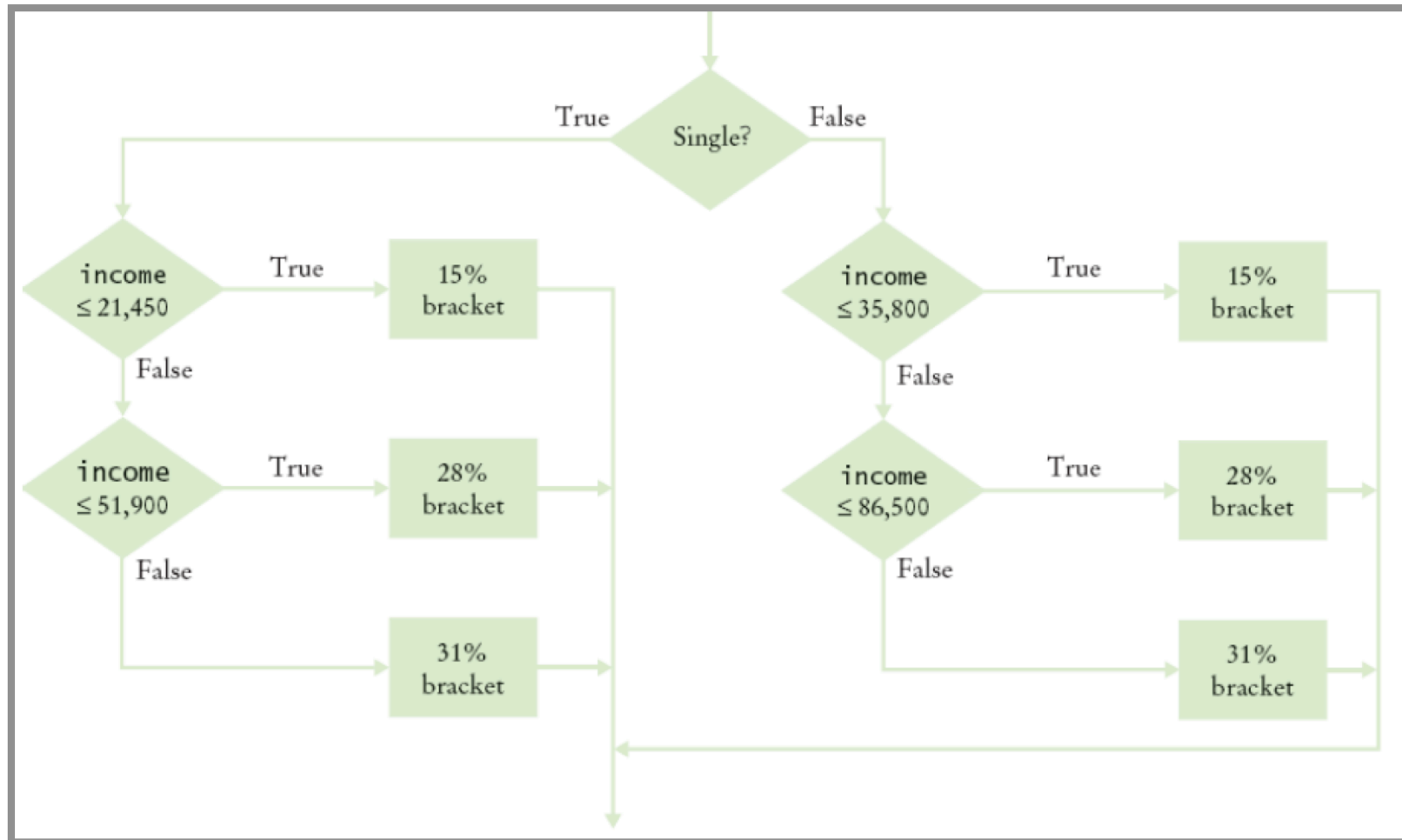


# Example: Computing Taxes

| If your filing status is single                 |            | If your filing status is married                |            |
|-------------------------------------------------|------------|-------------------------------------------------|------------|
| Tax Bracket                                     | Percentage | Tax Bracket                                     | Percentage |
| <b>\$0 ... \$21,450</b>                         | <b>15%</b> | <b>\$0 ... \$35,800</b>                         | <b>15%</b> |
| <b>Amount over \$21,451,<br/>up to \$51,900</b> | <b>28%</b> | <b>Amount over \$35,800,<br/>up to \$86,500</b> | <b>28%</b> |
| <b>Amount over \$51,900</b>                     | <b>31%</b> | <b>Amount over \$86,500</b>                     | <b>31%</b> |



# Taxes Flowchart





# Tax Program

- [TaxReturn.java](#)
  - [TaxReturnTester.java](#)
- 
- Beware 'Dangling else': pg 210





# Preparing Test Cases

- Test cases should achieve complete coverage of input possibilities
- Tax program
  - 2 filing possibilities
  - 3 tax brackets
  - = 6 possible combinations
- To test the program, select 6 valid inputs and at least 1 invalid input (negative income)



# Selection Operator

*condition* ? *value1* : *value2*

- Combines *values* to yield another value depending on *condition*
  - if construct combines *statements*

```
if (x >= 0) y = x; else y = -x;
y = x >= 0 ? x : -x;
```



# switch Statement

- Replaces sequence of if/else/else comparing single integer value against constant alternatives

```
int digit;
. . .
if (digit == 1)
 System.out.print("one");
else if (digit == 2)
 System.out.print("two");
else if (digit == 3)
 System.out.print("three");
. . .
else if (digit == 9)
 System.out.print("nine");
else
 System.out.print("error");
```

```
switch (digit) {
 case 1: System.out.print("one");
 break;
 case 2: System.out.print("two");
 break;
 case 3: System.out.print("three");
 break;
 . . .
 case 9: System.out.print("nine");
 break;
 default: System.out.print("error");
 break;
}
```



## switch Statement (cont.)

- Case values must be constants and must be integers, characters, or enumerated constants
  - Cannot be used with floating point, string, or objects
- Without **break** statements, execution ‘falls through’ to the next case until the end



# The boolean Type

- George Boole (1815-1864): pioneer in the study of logic
- Value of an expression like `amount < 100` is either `true` or `false`
- `boolean` type: one of these two truth values
  - Sometimes referred to as 0 and 1

```
double amount = 0;
boolean b = amount < 1000;
System.out.println(b);
```



# Boolean Operators

- Used to combine boolean expressions
  - `&&` — ‘and’
  - `||` — ‘or’ (to type |, use ‘shift’ key + ‘\’)
  - `!` — ‘not’
  - Also called *logical operators*
- `if ( 0 < amount && amount < 1000 ) . . .`
  - Both conditions must be satisfied
- `if ( input.equals("S") || input.equals("M") ) . . .`
  - At least one of the conditions must be satisfied



# Boolean Operators (cont.)

- `if ( !input.equals("S") ) . . .`
  - Inverts the condition – if input is *not* “S”
- Truth tables

| A     | B          | A && B |
|-------|------------|--------|
| True  | True       | True   |
| True  | False      | False  |
| False | <i>Any</i> | False  |

| A     | B          | A  B  |
|-------|------------|-------|
| True  | <i>Any</i> | True  |
| False | True       | True  |
| False | False      | False |

| A     | !A    |
|-------|-------|
| True  | False |
| False | True  |

- Expressions can be simplified using rules of Boolean algebra - e.g. see Topic 6.5 (pg 218)

# Boolean Operators: Lazy/Short-Circuit Evaluation



- `&&` and `||` operators computed from left to right; stop evaluation as soon as truth value can be determined
  - ‘and’: if first condition is *false*, skips the second
  - ‘or’: if first condition is *true*, skips the second

```
if (input != null && Integer.parseInt(input) > 0) . . .
```





# Predicate Methods

- Methods that return `boolean` value

```
public class BankAccount {
 . . .
 public boolean isOverdrawn() {
 return balance < 0;
 }
}
```

- Can be used in conditions

```
if (harrysChecking.isOverdrawn()) . . .
```



# Useful Predicate Methods

- Character class

- `isDigit`
- `isLetter`
- `isUpperCase`
- `isLowerCase`

```
if (Character.isUpperCase(ch)) . . .
```

- Scanner class: `hasNextInt`, `hasNextDouble`

```
if (in.hasNextInt()) n = in.nextInt();
```



# Boolean Variables

```
private boolean married;
```

- Can store a truth value, or the outcome of a condition expression
  - `married = input.equals( "M" );`
- Can be used in expressions
  - `if ( married ) . . . else . . .`
  - `if ( !married ) . . .`



# Boolean Variables: 'Flags'

- Sometimes also called 'flags'
- Think carefully about names of variables
  - maritalStatus vs. married
- Don't write tests like this:
  - `if ( married == true ) . . . // Don't`
  - `if ( married == false ) . . . // Don't`
- Use this instead:
  - `if ( married ) . . .`
  - `if ( !married ) . . .`



# Artificial Intelligence

- Serious research: mid-1950s
- Successes?
  - Chess
  - Theorem-proving
  - OCR
- Failures?
  - Translation
  - Grammar-checking
- Most 'AI' techniques don't actually imitate human thinking
- Ethical issues? . . .