Multiple Conditions; Loops with Classes

Principles of Computer Programming I

Spring/Fall 20XX

AUGUSTA UNIVERSITY

Outline

- While loops with multiple conditions
- Classes with Booleans
- Input validation with objects
- While loops in methods

Multiple Conditions

- Loop *continues* if loop condition is `true` and *stops* if loop condition is `false`
- Combine 2 conditions with && (AND) when **both** must be true for loop to continue
  - Loop will stop if either condition is false
- Combine 2 conditions with || (OR) when **either** can be true and the loop should continue
  - Loop will stop only when both conditions are false

User Input Validation

- "Ask the user to enter a non-negative price, and repeat until they enter a valid value"
- Loop should stop when input is *valid*, therefore continue when input is *invalid*
- Condition 1: User input is not a number
- Condition 2: User's number is negative
- Combination: OR, since either one by itself means invalid input

User Input Validation

- Checking only condition 1:

```
decimal price;
bool success;
do
{
  Console.WriteLine("Please enter"
    + " a price.");
  success = decimal.TryParse(
    Console.ReadLine(), out price);
} while(!success);
```

true if user entered a number

true if user did *not* enter a number

- Checking only condition 2:

```
decimal price;
bool success;
do
{
  Console.WriteLine("Please enter"
    + " a price.");
  success = decimal.TryParse(
    Console.ReadLine(), out price);
} while(price < 0);
```

true if user entered a negative (invalid) number

Combining with ||

- "Ask the user to enter a non-negative price, and repeat until they enter a valid value"

```
decimal price;
bool parseSuccess;
do
{
  Console.WriteLine("Please enter a price "
    + "(must be non-negative).");
  parseSuccess = decimal.TryParse(Console.ReadLine(), out price);
} while(!parseSuccess || price < 0);
Item myItem = new Item(desc, price);
```

Combine with OR: Either condition means the loop should continue

Condition 1: User enters a non-number

Condition 2: User enters a negative number

At this point, *both* conditions are `false`

Combining with &&

- "Give the user 3 tries to enter a valid integer. After the 3rd attempt, use a default value of 1"
- Condition 1: Continue if user input is not a number

```
!parseSuccess
```

- Condition 2: Continue if user has made *less* than 3 attempts

```
attempts < 3
```

- Combination: AND, since both must be true to loop again
  - If either condition is false, loop should stop

Combining with &&

```
int intVar, attempts = 0;
bool parseSuccess;
do
{
  Console.WriteLine("Please enter an integer");
  parseSuccess = int.TryParse(Console.ReadLine(), out intVar);
  attempts++;
} while(!parseSuccess && attempts < 3);
if(!parseSuccess && attempts == 3)
{
  Console.WriteLine("Using the default value 1");
  intVar = 1;
}
```

Counter variable

Condition 1

Combine conditions

Condition 2

At this point, *either* condition 1 or condition 2 is false (or both)

Test if loop ended because user failed 3rd attempt

Negating Conditions

- `while` condition is logical negation of when loop will stop: If loop should stop when `a == true`, write `while(!a)`
- Helpful rule: `!(a || b)` is the same as `!a && !b`
- Example: Loop should stop if user enters "yes" or "no"

Stop when this is true:	`input == "yes" || input == "no"`
Continue when this is true:	`!(input == "yes" || input == "no")`
Logical equivalent:	`input != "yes" && input != "no"`

Negating Conditions

- `while` condition is logical negation of when loop will stop: If loop should stop when `a == true`, write `while(!a)`
- Helpful rule: `!(a || b)` is the same as `!a && !b`
- Applied to input validation: Stop if user enters a valid integer, or if 3 attempts have been used up

Stop when this is true: `parseSuccess || attempts >= 3`

Continue when this is true: `!parseSuccess && attempts < 3`

Combining More Conditions

- "Ask the user to enter an integer between -100 and 100"
- Keep asking the user for input until:
  - Input is a number
  - Input is ≥ -100
  - Input is ≤ 100

All need to be true (use &&)

- Loop should *continue* if:
  - Input is not a number
  - Input is < -100
  - Input is > 100

Only 1 needs to be true (use ||)

Combining More Conditions

- Condition 1: Input not an integer `!isInteger`
- Condition 2: Input is < -100 `number < -100`
- Condition 3: Input is > 100 `number > 100`

```
int number;
bool isInteger;
do
{
  Console.WriteLine("Enter an integer between -100 and 100.");
  isInteger = int.TryParse(Console.ReadLine(), out number);
} while(!isInteger || number < -100 || number > 100);
```

Outline

- While loops with multiple conditions
- **Classes with Booleans**
- Input validation with objects
- While loops in methods

Classes and Decisions

- Review: Uses of Booleans in classes
  - Data validation in setter methods and properties
  - Data validation in constructors
  - Methods with Boolean parameters
  - Constructors with Boolean parameters
  - Boolean instance variables

```
public int Length
{
  get
  {
    return length;
  }
  set
  {
    if(value >= 0)
      length = value;
  }
}
```

Methods with Boolean Parameters

- Change behavior of method based on condition

In class Room:

```
public double ComputeArea(bool useMeters)
{
  if(useMeters)
    return length * width;
  else
    return GetLengthFeet() * GetWidthFeet();
}
```

- Argument can be any Boolean expression

```
Console.WriteLine("Compute area in feet (f) or meters (m)?");
char userChoice = char.Parse(Console.ReadLine());
Console.WriteLine($"Area: {myRoom.ComputeArea(userChoice != 'f')}");
```

true if the user entered ‘m’

Constructors with Boolean Parameters

- Constructors must all have the same name – can't use a different name for different "versions"
- Use parameters to indicate different behavior

```
public Room(double lengthP, double widthP, string nameP, bool meters)
```

Convert from feet to meters if `meters` is `false`

`true` if the other parameters are in meters, `false` if they are in feet

```
public WeatherForecast(double lowTemp, double highTemp, bool celsius)
```

Convert to Celsius if `celsius` is `false`

`true` if the other parameters are in Celsius, `false` if they are in Fahrenheit

Boolean Instance Variables

- Represent object state as 1 of 2 alternatives
- Example: `taxable` instance variable for `Item`

```
class Item
{
  private string description;
  private decimal price;
  private bool taxable;
...
  public bool IsTaxable()
  {
    return taxable;
  }
  public void SetTaxable(bool taxableP)
  {
    taxable = taxableP;
  }
}
```

Boolean Instance Variables

- Can be used in Main method like other object attributes

```
if(myItem.IsTaxable())
{
  Console.WriteLine("Sales tax will be added");
}
```

- Can be used within methods to provide "smarter" behavior

```
public decimal GetPrice()
{
  if(taxable)
    return price + (price * SALES_TAX);
  else
    return price;
}
```

Outline

- While loops with multiple conditions
- Classes with Booleans
- **Input validation with objects**
- While loops in methods

Input Validation with Constructors

- Use a loop to validate user input before constructing an object

```
int length, width;
bool isInt;
do
{
  Console.WriteLine("Enter a positive length");
  isInt = int.TryParse(Console.ReadLine(), out length);
} while(!isInt || length < 0);
do
{
  Console.WriteLine("Enter a positive width");
  isInt = int.TryParse(Console.ReadLine(), out width);
} while(!isInt || width < 0);
Rectangle myRectangle = new Rectangle(length, width);
```

Input Validation with Methods

- When using user input to call a method, ensure it makes sense
- Method may also do its own validation
- Changing the price of an Item:

```
bool isNumber;
decimal newPrice;
do
{
  Console.WriteLine($"Enter a new price for {myItem.GetDescription()}");
  isNumber = decimal.TryParse(Console.ReadLine(), out newPrice);
} while(!isNumber || newPrice < 0);
myItem.SetPrice(newPrice);
```

Check result of TryParse

Check if user's value is valid

SetPrice will ignore a price < 0, but won't prompt the user

Input Validation with Methods

- To call `ComputeArea`, input must be ‘f’ or ‘m’, and a valid `char`
- Don’t assume other values mean “use meters”

Desired outcome: input is ‘f’ or ‘m’ → `userChoice == 'f' || userChoice == 'm'`

TryParse succeeded → `validChar`

Loop condition: input is **incorrect**

```
bool validChar;
char userChoice;
do
{
  Console.WriteLine("Compute area in feet (f) or meters (m)?");
  validChar = char.TryParse(Console.ReadLine(), out userChoice);
} while(!validChar || !(userChoice == 'f' || userChoice == 'm'));
Console.WriteLine($"Area: {myRoom.ComputeArea(userChoice == 'm')}");
```

Could also write `userChoice != 'f' && userChoice != 'm'`

Outline

- While loops with multiple conditions
- Classes with Booleans
- Input validation with objects
- **While loops in methods**

Loops Inside Methods

- Input validation is a common task
- Instead of repeating code, use a method

Inside class `Rectangle`:

```
public void SetLengthFromUser()
{
  bool isInt;
  do
  {
    Console.WriteLine("Enter a positive length");
    isInt = int.TryParse(Console.ReadLine(), out length);
  } while(!isInt || length < 0);
}
```

No parameter, gets input from user

Assign output to instance variable

Loops Inside Methods

- Loop can depend on object attributes
- Example: "Draw" a rectangle using symbols in the console

```
public void DrawInConsole()
{
  int counter = 1;
  while(counter <= width * length)
  {
    Console.Write(" * ");
    if(counter % width == 0)
      Console.WriteLine();
    counter++;
  }
}
```

Initial value must be 1, not 0

Total number of * to draw: Area of rectangle

Each line should end after printing `width` asterisks

Loops Using Methods

- Methods can return Boolean values
- Other code can use return value to control a loop
- Example: Comparing `Time` objects
  - We already wrote `GetTotalSeconds()`
  - Define `ComesBefore` method to decide if this `Time` object comes before another

```
public bool ComesBefore(Time otherTime)
{
  return GetTotalSeconds() < otherTime.GetTotalSeconds();
}
```

Loops Using Methods

- Ask the user to enter a `Time` less than a certain maximum
  - Use user input to construct a `Time` object
  - Use `ComesBefore` to determine whether to prompt user again

```
Time maximumTime = new Time(2, 45, 0);
Time userTime;
do
{
  //Prompt user for input of hours, minutes, and seconds
  userTime = new Time(hours, minutes, seconds);
} while(!userTime.ComesBefore(maximumTime));
```

Keep looping until `ComesBefore` returns `true`

```
Time maximumTime = new Time(2, 45, 0);
Time userTime;
do
{
  Console.WriteLine($"Enter a time less than {maximumTime}");
  int hours, minutes, seconds;
  do
  {
    Console.Write("Enter the hours: ");
  } while(!int.TryParse(Console.ReadLine(), out hours));
  do
  {
    Console.Write("Enter the minutes: ");
  } while(!int.TryParse(Console.ReadLine(), out minutes));
  do
  {
    Console.Write("Enter the seconds: ");
  } while(!int.TryParse(Console.ReadLine(), out seconds));
  userTime = new Time(hours, minutes, seconds);
} while(!userTime.ComesBefore(maximumTime));
```

Declare outside loop, initialize inside loop

ToString will be called, assuming we wrote it

Return value of TryParse can be used in loop condition

Create a new object each time