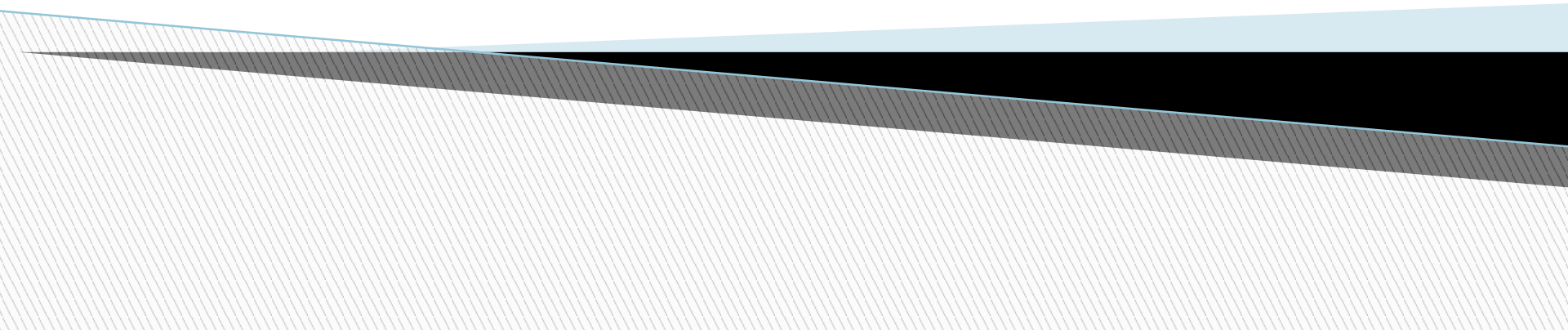


Readability & Design Considerations



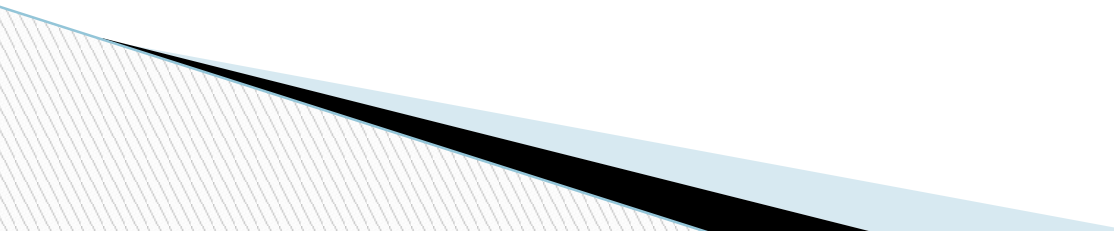
Today's Lecture

- Clean code
 - Commenting
 - Design Considerations
 - Functions
 - Classes
- 

Boy Scouts rule

“Developers should leave the code base cleaner than they found it . Just as boy scouts are taught to leave a campsite cleaner than they found it, software developers should strive to improve the codebase with every change they make“

Clean Code

- Simple and expressive
 - Well Organized
 - Readable and Understandable
 - Testable and reliable
 - Efficient and Optimized
- 

Commenting

Purpose of commenting

- Who is your audience?
 - other developers
 - your future self
- Explain the intent of the code
- Clarify complex logic
- Provide context to future developers to modify or debug code

Other uses

- Document non obvious assumptions & constraints
- 

Good Comment

A comment that clarifies complex logic, explains design decisions or tradeoffs , documents non obvious requirements.

Good comments provide relevant information that cannot be inferred from the code itself

Explanation of Intent

A comment that explains the intent of code, its purpose and its expected behaviour

// Calculates the sum of all elements in the array.

```
int sum = 0;
for (int i = 0; i < arr.length; i++) {
    sum += arr[i];
}
```

Clarification of Code

A Comment that clarifies complex logic such as a tricky conditional or a loop

// Loop through all elements in the array, but skip any that are null or empty.

```
for (int i = 0; i < arr.length; i++) {  
    if (arr[i] == null || arr[i].isEmpty()) {  
        continue;  
    }  
    // Process the non-null, non-empty element here.  
}
```


Warning of Consequences

A comment that warns of potential side effects or consequences of using the code*

// Be careful when calling this method with large inputs, as it can cause a stack overflow.

```
public int recursiveFunction(int n) {  
    if (n <= 0) {  
        return 0;  
    }  
    return n + recursiveFunction(n - 1);  
}
```

*where possible, be more specific about what is considered “large”

Marker in Code

A comment that highlights areas of code that require further work or improvements

```
// TODO: Add error handling for invalid inputs.
```

```
public int divide(int x, int y) {  
    return x / y;  
}
```

Summary of Code

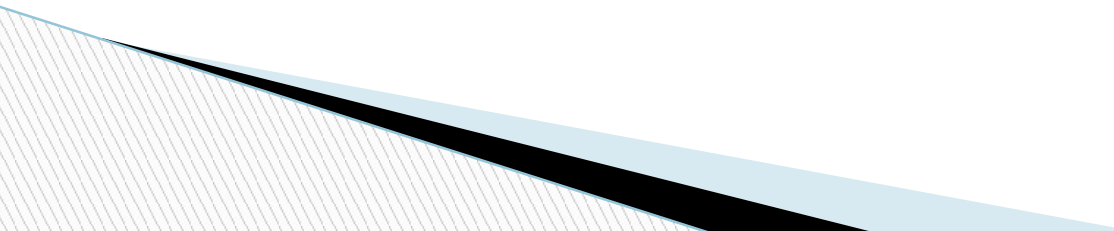
Summary comments convey the meaning of the entire code at a glance so developers do not have to read the entire code block

```
/**  
 * This function calculates the distance between two points on a 2D plane  
 * using the Pythagorean theorem:  $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ .  
 * @param x1 the x-coordinate of the first point  
 * @param y1 the y-coordinate of the first point  
 * @param x2 the x-coordinate of the second point  
 * @param y2 the y-coordinate of the second point  
 * @return the distance between the two points  
 */  
public double calculateDistance(double x1, double y1, double x2, double y2) {  
    double xDiff = x2 - x1;  
    double yDiff = y2 - y1;  
    double sumOfSquares = (xDiff * xDiff) + (yDiff * yDiff);  
    return Math.sqrt(sumOfSquares);  
}
```

Bad Comment

A comment that reiterates the code, contains outdated or incorrect information*, explains obvious or trivial code , or adds irrelevant or inappropriate content.

* while it's important to keep comments up-to-date, it's common that this is forgotten. Avoid comments that can easily become irrelevant if left unmodified when code changes.



Redundant Comments

Comments that simply repeat the code in a slightly different way

```
// Add one to the value of x  
x = x + 1;
```

Too many comments of this kind results in “comment pollution” which detracts from the quality of the code base

Misleading Comments

Comments that are incorrect or no longer true

```
// This function is optimized for speed  
// and doesn't need any error checking
```

```
function process_data(data) {  
    // do something with data  
}
```

A better comment would ask the user to call the function with data that is already pre validated

Journal Comments

When developers use comments instead of git for version control of the file

```
// 2002-07-20 - Bob: Changed the calculation of the tax rate to account for
//           the new tax laws in Maine. Taxes should now be
//           calculated correctly for all residents.
// 2002-06-19 - Rob: Created the function to calculate the tax rate to account
//
```

Position Markers

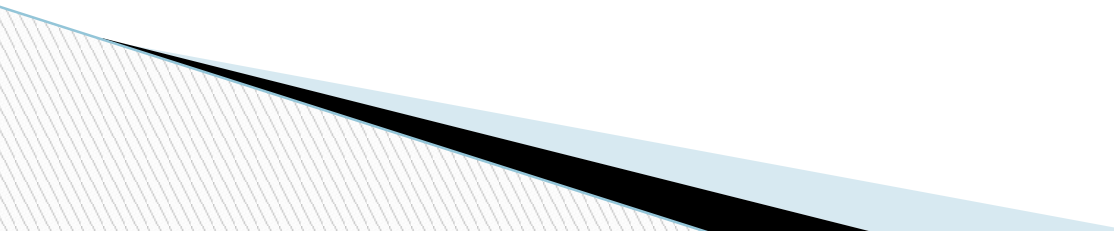
Comments used to separate sections of code which can be usually achieved through better organization

```
// ***** BEGIN MAIN METHOD *****  
public static void main(String[] args) {  
    // code here  
}  
// ***** END MAIN METHOD *****
```


Comment the Why, not the How

Example from “recover.c”

```
if (track >= 1024)
    printf("%i WARNING (cylinder) in
    %s\n",status,filename);
else
    write_count =
    write(outFile,io_buff,bytes_2_write);
```



Comment the Why, not the How

Example from “recover.c”

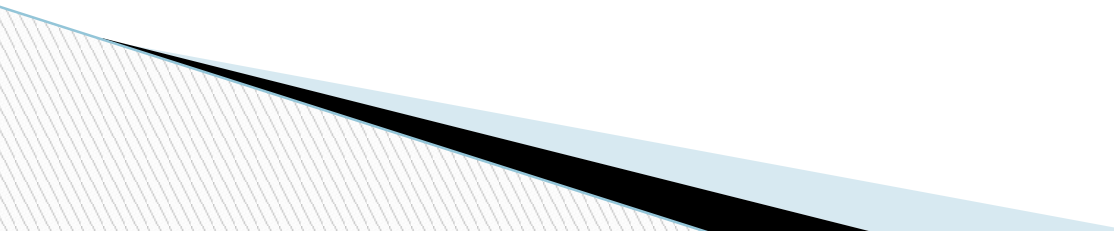
```
// this BIOS can not read tracks greater than 1024 without special drivers
// print warning but keep recovering the file anyway
if (track >= LAST_LEGAL_TRACK)
    printf("%i WARNING (cylinder) in %s\n",status,filename);
else
    write_count = write(outFile,io_buff,bytes_2_write);
```

***notice the literal or magic number “1024” replaced with “LAST_LEGAL_TRACK”**

Commenting Routines

- Say what the routine WON'T do, mention permissible input values
- Document global effects (if any)
- Side effects (are dangerous)
- Create or destroy anything?
- Document source of algorithms
- Avoid enormous comment blocks
 - I like some comments before every routine for visual delineation at the very least

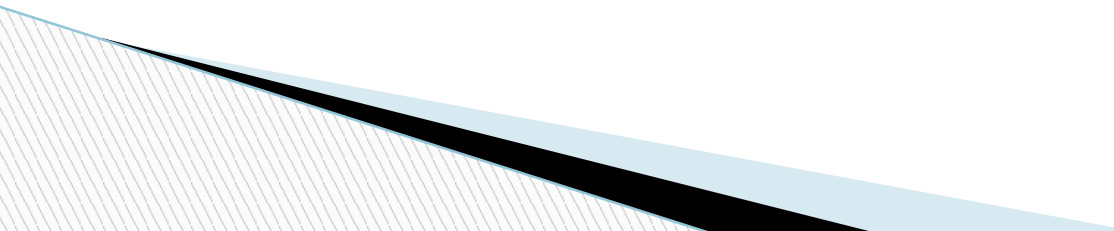
Balance

- Certainly need a useful amount of commenting
 - Avoid “comment pollution”
 - If it takes too much time to wade through comments then there’s too much
 - If there’s as much or more comments than code then there’s too much
 - Rule of thumb: 1 line of comment for 10 lines of code
 - Do not count lines and then add comments every 10th line!
- 

Design Considerations



Design at the Function level

- Functions should be small, do one thing and do it well
 - Should have descriptive names that accurately reflect their purpose
 - Limited number of arguments:
 - ideally 0 or one.
 - More than 3 makes them difficult to understand
 - seven is considered the upper limit (McConnell)
 - Should have minimal side effects , meaning they do not modify any state outside of their scope
 - Should follow single responsibility principle, they should have only one reason to change
 - Organized in a logical order with higher level functions calling lower level functions
 - Avoid using flag arguments (Catch all functions)
- 

Do only one thing

Functions should be small, do one thing and do it well

// Bad Example

```
function calculateAndPrintInvoice() {  
  // A long list of calculations and printing statements  
}
```

// Good Example

```
function calculateInvoice() {  
  // A short list of calculations  
  return invoice;  
}
```

```
function printInvoice(invoice) {  
  // A short list of printing statements  
}
```

Minimal number of arguments

Limited number of arguments ideally 0 or one. More than 3 makes them difficult to understand

// Bad Example

```
function createPerson(firstName, lastName, age, streetAddress, city, state, zipCode) {  
  // Creating person object with all arguments  
}
```

// Good Example

```
function createPerson(personData) {  
  // Creating person object with personData object  
}
```


Avoid side effects

Should have minimal side effects , meaning they do not modify any state outside of their scope

// Bad Example

```
function updateAge() {  
  user.age = user.age + 1;  
}
```

// Good Example

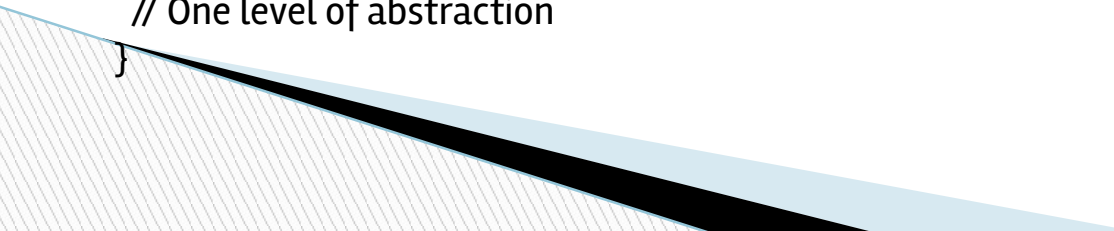
```
function getNextAge(age) {  
  return age + 1;  
}
```

One Level of Abstraction

Organized in a logical order with higher level functions calling lower level functions

```
// Bad Example
function doSomethingComplicated() {
  // A long list of steps, each one with multiple levels of abstraction
}
```

```
// Good Example
function doSomethingComplicated() {
  stepOne();
  stepTwo();
}
function stepOne() {
  // One level of abstraction
}
function stepTwo() {
  // One level of abstraction
}
```



Avoid catch all functions

Organized in a logical order with higher level functions calling lower level functions

```
// Bad Example
function calculateShippingCosts(order) {
  switch(order.shippingMethod) {
    case 'UPS':
      // Calculate UPS shipping costs
      break;
    case 'USPS':
      // Calculate USPS shipping costs
      break;
    case 'FedEx':
      // Calculate FedEx shipping costs
      break;
    default:
      // Throw an error
  }
}
```

```
// Good Example
function calculateShippingCostsUPS(order) {
  // Calculate UPS shipping costs
}

function calculateShippingCostsUSPS(order) {
  // Calculate USPS shipping costs
}

function calculateShippingCostsFedEx(order) {
  // Calculate FedEx shipping costs
}
```

Design at the class level

- Your designs must consist of highly cohesive, loosely coupled components (e.g. your design is highly normalized) to make testing easier (this also makes evolution and maintenance of your system easier too).

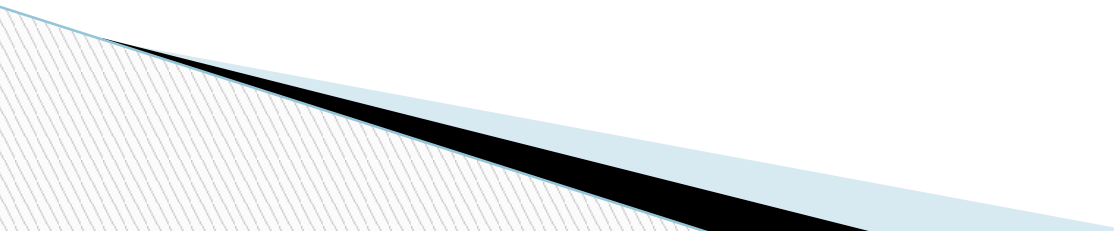
(Beck)



Don't put off until tomorrow...

it's not uncommon to “put off” exception handling and error handling.

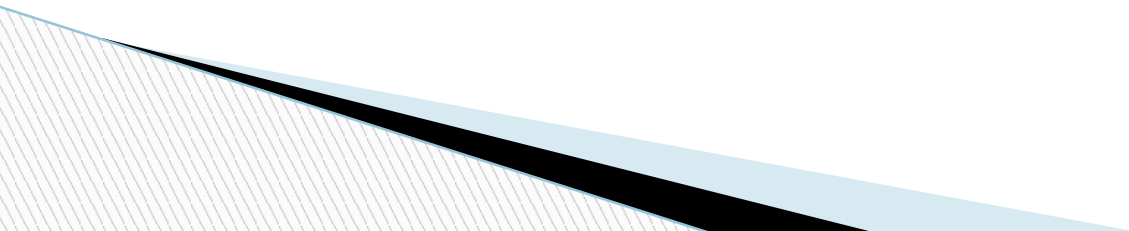
Front load your development. Spend the time early to avoid bigger hassles later. Skipping these things usually causes more work at a later time than doing it right early on.



Cohesion and Coupling

Cohesion is a measure of how strongly-related and focused the responsibilities of a single class are

Coupling is the degree to which each program module relies on each one of the other modules



High Cohesion

```
public class Stack {
    private int[] elements;
    private int top = -1;

    public Stack(int maxSize) {
        elements = new int[maxSize];
    }

    public void push(int element) {
        if (top >= elements.length - 1) {
            throw new StackOverflowException();
        }
        elements[++top] = element;
    }

    public int pop() {
        if (top < 0) {
            throw new StackUnderflowException();
        }
        return elements[top--];
    }
}
```

The Stack class is highly cohesive because it has a clear and focused responsibility – managing a stack of elements. All of its methods are related to this responsibility and work together to achieve it.

Low Cohesion

```
public class DataProcessor {  
    public void readDataFromFile(String fileName) {  
        // code to read data from file  
    }  
  
    public void processData() {  
        // code to process data  
    }  
  
    public void writeDataToFile(String fileName) {  
        // code to write data to file  
    }  
  
    public void sendDataToServer(String serverUrl) {  
        // code to send data to server  
    }  
}
```

DataProcessor class is loosely cohesive because it has multiple responsibilities that are not closely related to each other. It has methods for reading data from a file, processing the data, writing the data to a file, and sending the data to a server. These responsibilities could be split up into separate classes that each have a single, well-defined responsibility.

Tight Coupling

```
public class OrderProcessor {  
    private InventorySystem inventory;  
    private BillingSystem billing;  
  
    public OrderProcessor() {  
        inventory = new InventorySystem();  
        billing = new BillingSystem();  
    }  
  
    public void processOrder(Order order) {  
        // Update inventory  
        inventory.update(order);  
  
        // Charge customer  
        billing.charge(order);  
    }  
}
```

The OrderProcessor class is highly coupled with the InventorySystem and BillingSystem classes. It directly creates instances of these classes and calls methods on them to process orders. This makes it difficult to modify or test the OrderProcessor class without also modifying or testing the InventorySystem and BillingSystem classes.

Loose Coupling

```
public interface IOrderProcessor {  
    void processOrder(Order order);  
}
```

```
public class InventorySystem implements IOrderProcessor {  
    public void processOrder(Order order) {  
        // Update inventory  
    }  
}
```

```
public class BillingSystem implements IOrderProcessor {  
    public void processOrder(Order order) {  
        // Charge customer  
    }  
}
```

```
public class OrderProcessor {  
    private IOrderProcessor inventory;  
    private IOrderProcessor billing;
```

```
    public OrderProcessor(IOrderProcessor  
inventory, IOrderProcessor billing) {  
        this.inventory = inventory;  
        this.billing = billing;  
    }
```

```
    public void processOrder(Order order) {  
        // Update inventory  
        inventory.processOrder(order);  
  
        // Charge customer  
        billing.processOrder(order);  
    }  
}
```

the OrderProcessor class is loosely coupled with the InventorySystem and BillingSystem classes. It depends only on the IOrderProcessor interface, which is implemented by the InventorySystem and BillingSystem classes. This allows for easier modification and testing of the OrderProcessor class, as well as easier substitution of different implementations of the IOrderProcessor interface.

General Rule of Thumb

“The biggest piece of code is small enough that you can describe, comment, write and test without thinking much about how to do it” – M.Woodley

Following the principles discussed in this lecture will help you write code according to this rule