Day 2

COMP 1006/1406A

Summer 2016

M. Jason Hinek
Carleton University

today's agenda

- a quick look back (Monday's class)
- assignments
  - a1 is due on Monday
  - a2 will be available on Monday and is due the following Monday
- testing
  - black box and white box testing
- problem solving
- More Java
  - Classes/Objects

last time...

- primitive data types
  - `byte`, `short`, `int`, `long` (integers)
  - `float`, `double` (approximate real numbers)
  - `boolean` (logical true/false)
  - `char` (unicode characters)
- type conversions
  - automatic, cast operators `(type)`, methods
- order of operations
  - for example,
    array access `[ ]` > unary negation `-` > cast `( )` > multiplication `*`

last time...

- primitive data types
  - `byte`, `short`, `int`, `long` (integers)
  - `float`, `double` (approximate real numbers)
  - `boolean` (logical true/false)
  - `char` (unicode characters)
- type conversions
  - automatic, cast operators `(type)`, methods
- order of operations
  - for example,
    array access `[ ]` > unary negation `-` > cast `( )` > multiplication `*`

aside: what do the following lines do?

```
boolean b = false;
b || true;
if( b = true ) System.out.println("hi");
```

last time...

- reference data types (objects)
  - everything that isn't a primitive data type
  - `new` operator allocates memory for objects
- memory model
  - primitive data type variables store actual values (data)
  - everythign else stores reference to actual object (or `null`)
  - it gets messy...

last time...

```java
/* hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- access modifier `public`
  - top level access modifier specifies who can see `HelloWorld`
  - member level access modifier specifies who can access `main`
- (non access) modifier `static`
  - allows a method to be called without an instance of the class
- return type `void`
  - it is a Java keyword that tells us that a method returns nothing
  - it is not an actual Java type
- `System.out`
  - `System` is a class with three attributes/fields `in`, `out` and `err`
  - `out` is a `PrintStream` object, it is “standard output”
  - `println` is a method of `out`

last time...

```
/* hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- what do we know about `System.out.println()`?

last time...

```
/* hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- what do we know about `System.out.println()`?
- we can look at the API for the System class <link>
- API - application programming interface
    - specifies how to use a given class

running hello world

```
/* Java hello world */
public class HelloWorld{
  public static void main(String[] args){
    System.out.println("hello, world!");
  }
}
```

- Java convention is that
  - class name is capitalized (use camel case if more than one word)
  - class `XXX` must be in the file `XXX.java`
  - so `HellowWorld` must be in the file `HelloWorld.java`
- first we need to compile the source code into Java bytecode
  - IDE will have a compile button
  - `javac HelloWorld.java` from console window (shell)
  - this creates `HelloWorld.class`, which is the Java bytecode
- next, we run the bytecode in the JVM (Java virtual machine)
  - `java HelloWorld` from the console window runs out program!
  - the JVM executes the `main` method of our program

What we didn't get to...

Abstraction

> *In software engineering and computer science,* `abstraction` *is a technique for managing complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level.*

-wikipedia

Abstraction

> *In software engineering and computer science, `abstraction` is a technique for managing complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level.*

-wikipedia

hardware

Abstraction

> *In software engineering and computer science, `abstraction` is a technique for managing complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level.*

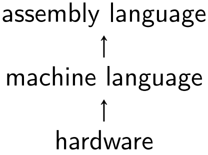
-wikipedia

machine language
↑
hardware

Abstraction

> *In software engineering and computer science, `abstraction` is a technique for managing complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level.*

-wikipedia



assembly language

↑

machine language

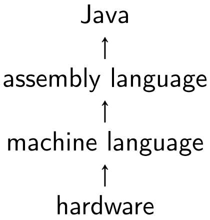
↑

hardware

Abstraction

> *In software engineering and computer science,* `abstraction` *is a technique for managing complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level.*

-wikipedia



Java

↑

assembly language

↑

machine language

↑

hardware

Abstraction

A first step in abstraction for us is the notion of a procedure

- procedure
- function
- method
- subroutine

A procedure is a sequence of instructions (statements) that performs a specific task.

Abstraction

A first step in abstraction for us is the notion of a procedure

- procedure
- function
- method
- subroutine

A procedure is a sequence of instructions (statements) that performs a specific task.

`Math.sin(0.172)` computes the sine of 0.172 radians. We don't really care *how* it does it, we just care that it gives us the right value.

`System.out.println("hi")` prints the characters h and i to the screen. How does this actually happen? Do we care how it happens?

Procedural programming

- program is a sequence of function calls
- helps us write structured programs
  - this helps `us` write better programs (Software Engineering)
- code that does one thing is grouped into a procedure (function/method)
  - helps us achieve modularization
- code that is repeated during the execution of the program is put in a procedure (function/method)
  - lets us write less code (less testing and maintenance!))

You have already been doing this in COMP1005/1405.

Procedural programming

```
public class Game{
  static int updateGame(int lives){...}
  static boolean checkContinue(){...}
  static void updateGraphics(){..}

  public static void main(String[] args){
    int lives = 3;
    boolean keepPlaying = true;
    while(lives > 0 && keepPlaying){
      lives = updateGame(lives);
      keepPlaying = checkContinue();
      updateGraphics();
    }
  }
}
```

Procedural programming

In Java, procedural programming involves writing and using a collection of `static` functions

You will most likely still use objects when writing code in this style in Java

- arrays are objects
- strings are objects

command line arguments vs user input

both are ways of providing some information to your program

- command line arguments
  - input is entered before program runs
  - input is passed as parameters to `main` method (`args`)
  - (not practical for many user inputs)
  - ((very useful for testing))
- user input
  - input is entered while program is running
  - standard input is the keyboard
  - (very flexible and useful)

Some Java classes

Let's take a quick look at some classes that Java provides

`https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html`
`https://docs.oracle.com/javase/8/docs/api/java/lang/String.html`
`https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html`

Anything found in `java.lang` is always visible to your program.
Using scanner we need to `import` the class.

Some Java classes

Let's take a quick look at some classes that Java provides

`https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html`
`https://docs.oracle.com/javase/8/docs/api/java/lang/String.html`
`https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html`

Anything found in `java.lang` is always visible to your program.
Using scanner we need to `import` the class.

let’s take a break...
for 5 minutes

testing

Testing

how do you know if your code is correct?

Testing

how do you know if your code is correct?

- it compiled ✓

Testing

how do you know if your code is correct?

- it compiled ✓
- it ran without an error ✓

Testing

how do you know if your code is correct?

- it compiled ✓
- it ran without an error ✓
- it gave the same result as the example in the assignment ✓

Testing

how do you know if your code is correct?

- it compiled ✓

- it ran without an error ✓

- it gave the same result as the example in the assignment ✓

what does this really mean though?

Testing

how do you know if your code is correct?

- ~~it compiled ✓~~ ✗

- ~~it ran without an error ✓~~ ✗

- ~~it gave the same result as the example in the assignment ✓~~ ✗

what does this really mean though?

Testing

how do you know if your code is correct?

- ~~it compiled ✓~~ ✗
  if it did not compile it is not correct code
- ~~it ran without an error ✓~~ ✗
- ~~it gave the same result as the example in the assignment ✓~~ ✗

what does this really mean though?

Testing

how do you know if your code is correct?

- ~~it compiled ✓~~ ✗
  if it did not compile it is not correct code
- ~~it ran without an error ✓~~ ✗
  if there was a runtime error it is not correct/robust code
- ~~it gave the same result as the example in the assignment ✓~~ ✗

what does this really mean though?

Testing

how do you know if your code is correct?

- ~~it compiled ✓~~ ✗
  if it did not compile it is not correct code
- ~~it ran without an error ✓~~ ✗
  if there was a runtime error it is not correct/robust code
- ~~it gave the same result as the example in the assignment ✓~~ ✗
  if the output is different from assignment example

what does this really mean though?

Testing

how do you know if your code is correct?

- ~~it compiled ✓~~ ✗
  if it did not compile it is not correct code
- ~~it ran without an error ✓~~ ✗
  if there was a runtime error it is not correct/robust code
- ~~it gave the same result as the example in the assignment ✓~~ ✗
  if the output is different from assignment example ???

what does this really mean though?

Testing

how do you know if your code is correct?

Testing

how do you know if your code is correct?

- need to try all possible inputs to program

Testing

how do you know if your code is correct?

- need to try all possible inputs to program
  - consider input of a single `int` (32-bits)
  - how many possible inputs are there?

Testing

how do you know if your code is correct?

- need to try all possible inputs to program
  - consider input of a single `int` (32-bits)
  - how many possible inputs are there? (4,294,967,296)

Testing

how do you know if your code is correct?

- need to try all possible inputs to program
  - consider input of a single `int` (32-bits)
  - how many possible inputs are there? (4,294,967,296)
  - what happens when there are two integers?

Testing

how do you know if your code is correct?

- need to try all possible inputs to program
  - consider input of a single `int` (32-bits)
  - how many possible inputs are there? (4,294,967,296)
  - what happens when there are two integers? (18,446,744,073,709,551,616)

Testing

how do you know if your code is correct?

- need to try all possible inputs to program
  - consider input of a single `int` (32-bits)
  - how many possible inputs are there? (4,294,967,296)
  - what happens when there are two integers? (18,446,744,073,709,551,616)
  - in general, this is not feasible for even simple programs...

Testing

how do you know if your code is correct?

- need to try all possible inputs to program
  - consider input of a single `int` (32-bits)
  - how many possible inputs are there? (4,294,967,296)
  - what happens when there are two integers? (18,446,744,073,709,551,616)
  - in general, this is not feasible for even simple programs...
- "prove" that it is correct

Testing

how do you know if your code is correct?

- need to try all possible inputs to program
  - consider input of a single `int` (32-bits)
  - how many possible inputs are there? (4,294,967,296)
  - what happens when there are two integers? (18,446,744,073,709,551,616)
  - in general, this is not feasible for even simple programs...
- "prove" that it is correct
  - formal methods is an entire research field devoted to this
  - way beyond the scope of this course!

Testing

how do you know if your code is correct?

- need to try all possible inputs to program
    - consider input of a single `int` (32-bits)
    - how many possible inputs are there? (4,294,967,296)
    - what happens when there are two integers? (18,446,744,073,709,551,616)
    - in general, this is not feasible for even simple programs...
- "prove" that it is correct
    - formal methods is an entire research field devoted to this
    - way beyond the scope of this course!
- in reality, we just don't know

Testing

how do you know if your code is correct?

- need to try all possible inputs to program
  - consider input of a single `int` (32-bits)
  - how many possible inputs are there? (4,294,967,296)
  - what happens when there are two integers? (18,446,744,073,709,551,616)
  - in general, this is not feasible for even simple programs...
- "prove" that it is correct
  - formal methods is an entire research field devoted to this
  - way beyond the scope of this course!
- in reality, we just don't know
  - we can have confidence that the code is relatively bug free
  - test, test and test...

Testing

how do you know if your code is correct?

- need to try all possible inputs to program
  - consider input of a single `int` (32-bits)
  - how many possible inputs are there? (4,294,967,296)
  - what happens when there are two integers? (18,446,744,073,709,551,616)
  - in general, this is not feasible for even simple programs...
- "prove" that it is correct
  - formal methods is an entire research field devoted to this
  - way beyond the scope of this course!
- in reality, we just don't know
  - we can have confidence that the code is relatively bug free
  - test, test and test...
    - but let's put some thought into the tests!
    - reduce the number of redundant tests!
    - only use "good" tests

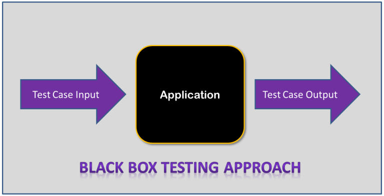
Testing

consider two types of testing

Testing

consider two types of testing

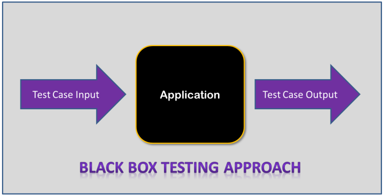
- black box testing



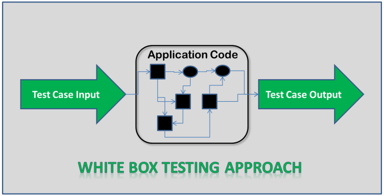
Testing

consider two types of testing

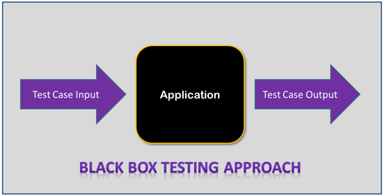
- black box testing



- white box testing (clear box testing)

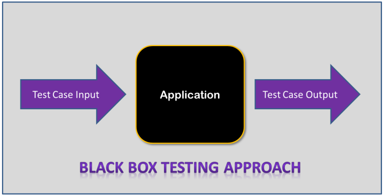


Black box testing



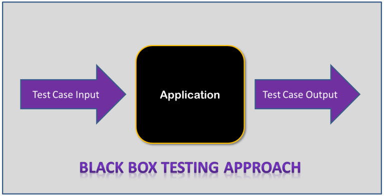
- no details of data structures known
- no details of algorithms used
- but, we do have access to the **interface** (API for Java classes)

Black box testing



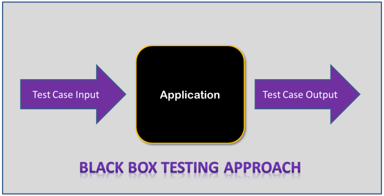
- no details of data structures known
- no details of algorithms used
- but, we do have access to the **interface** (API for Java classes)
    - interface is like a contract between code writer and code user

Black box testing



- no details of data structures known
- no details of algorithms used
- but, we do have access to the **interface** (API for Java classes)
  - interface is like a contract between code writer and code user
  - we have contracts for methods in class

Black box testing



- no details of data structures known
- no details of algorithms used
- but, we do have access to the **interface** (API for Java classes)
  - interface is like a contract between code writer and code user
  - we have contracts for methods in class
  - will construct test cases based on the contracts

Black box testing

```
boolean greaterThan10(int n):
    greaterThan10 :  int -> boolean
    purpose:  determine if n is greater than 10
    preconditions:
        n is a non-negative integer
    postconditions:
        outputs true if n > 10
        outputs false if n <= 10
```

Black box testing

```
boolean greaterThan10(int n):
    greaterThan10 :  int -> boolean
    purpose:  determine if n is greater than 10
    preconditions:
        n is a non-negative integer
    postconditions:
        outputs true if n > 10
        outputs false if n <= 10
```

- what do the pre- and post-conditions tell us?

Black box testing

```
boolean greaterThan10(int n):
    greaterThan10 :  int -> boolean
    purpose:  determine if n is greater than 10
    preconditions:
        n is a non-negative integer
    postconditions:
        outputs true if n > 10
        outputs false if n <= 10
```

- what do the pre- and post-conditions tell us?

- special cases?

Black box testing

```
boolean greaterThan10million(int n):
    greaterThan10million :  int -> boolean
    purpose:  determine if n is greater than 10 milion
    preconditions:
        n is a non-negative integer
    postconditions:
        outputs true if n > 10 million
        outputs false if n <= 10 million
```

- what do the pre- and post-conditions tell us?
- special cases?

Black box testing

```
boolean greater(int a, int b):
    greater :  int int -> boolean
    purpose:  determine if a is greater than b
    preconditions:
        a is a non-negative integer
        b is an integer
    postconditions:
        outputs true if a > b
        outputs false if a <= b
```

Black box testing

```
boolean greater(int a, int b):
    greater :  int int -> boolean
    purpose:  determine if a is greater than b
    preconditions:
        a is a non-negative integer
        b is an integer
    postconditions:
        outputs true if a > b
        outputs false if a <= b
```

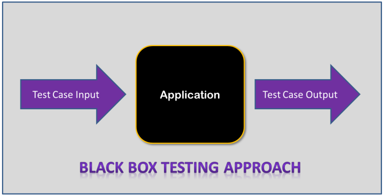
- what do the pre- and postconditions tell us?

Black box testing

```
boolean greater(int a, int b):
    greater :  int int -> boolean
    purpose:  determine if a is greater than b
    preconditions:
        a is a non-negative integer
        b is an integer
    postconditions:
        outputs true if a > b
        outputs false if a <= b
```

- what do the pre- and postconditions tell us?
- special cases?

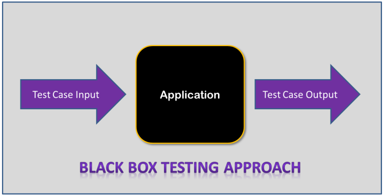
Black box testing



Test the contract

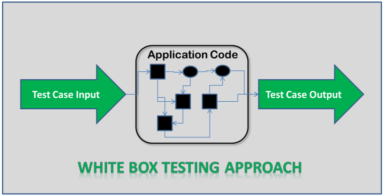
- boundary cases
- near-boundary cases (off-by-one error?)
- extreme cases (does the program scale?)
- special cases (program dependent, all of the above)
- typical/average cases

Black box testing



- test each method individually
- test methods interacting with each other
- test entire program

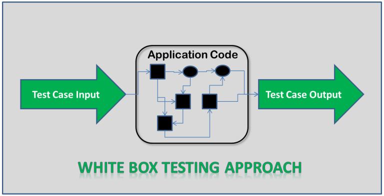
White box testing



Test the code

- we have knowledge of data structures used
- we have details of algorithms used
- we may have the actual code

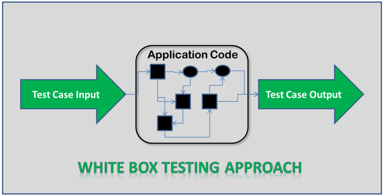
White box testing



Test the code

- we have knowledge of data structures used
- we have details of algorithms used
- we may have the actual code (assume this)
    - test each branch in the method / program

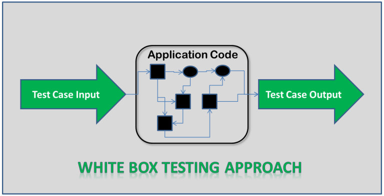
White box testing



Test the code

- we have knowledge of data structures used
- we have details of algorithms used
- we may have the actual code (assume this)
  - test each branch in the method / program
  - have each line of code in at least one test

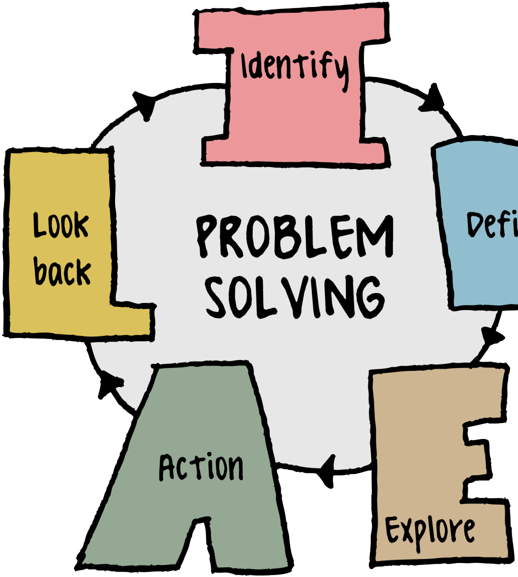
White box testing



Test the code

- we have knowledge of data structures used
- we have details of algorithms used
- we may have the actual code (assume this)
    - test each branch in the method / program
    - have each line of code in at least one test
    - use same approach as black box testing

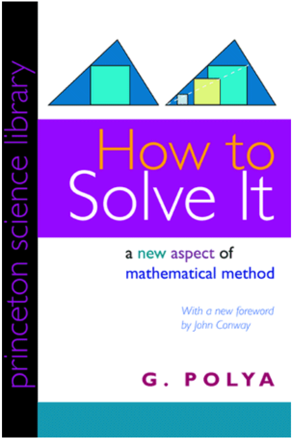
let’s take a break...
for 5 minutes


Identify
Define
Explore
Action
Look back
PROBLEM SOLVING

Problem Solving

George Pólya

- How to Solve it
- Terminology: data, unknown, condition


princeton science library
How to Solve It
a new aspect of mathematical method
With a new foreword by John Conway
G. POLYA

Problem Solving

Using the given **data** to find the **unknown** such that the **condition** is satisfied.

- the data is the information you have.
- the unknown is the information you want.
- the condition is the constraints on the problem.
  These are rules (often implicit) that must be followed.

Alternatively...

Using the given **data** to achieve a **goal** such that the **condition** is satisfied.

Using the given **data** to create an **algorithm/program** that achieves a **goal** such that the **constraints** are satisfied.

Problem Solving

The four phases of problem solving

1. Understand the problem.
    - identify the data/unknown/condition
2. Devise a plan.
    - choose a technique/heuristic/approach
    - start over if needed
3. Carry out the plan.
    - execute your plan
    - check each step
    - start over if needed
4. Look back.
    - reflect on what you did
    - start over if needed

Problem Solving

General strategies

- Related problems
    - transform the problem into one you already know how to solve
- Abstraction
    - remove details that are not relevant to the problem
- Divide and Conquer
    - break the problem into (smaller) sub-problems
- Backward Chaining
    - start from the solution and work backwards

Problem Solving

General strategies (from Think Like a Programmer, V. A. Spraul)

- Always have a plan
- Restate the problem
- Break the problem down
- Start with what you know
- Reduce the problem
- Look for analogies
- Experiment
- Don't get frustrated!

Problem Solving

General strategies (from Think Like a Programmer, V. A. Spraul)

- Always have a plan
  - Aimless wandering wastes time.
  - Without a plan, you are hoping for a lucky break.
  - Plans give you intermediate goals.
  - Plans can change.
- Restate the problem
  - Check out the problem from every angle before starting.
  - We may find the goal is not what we thought.
  - Use restatement to confirm understanding.
- Break the problem down
  - Divide the problem into steps or phases.
  - Difficulty for each phase can be an order of magnitude lower.
  - Sometimes the sub-problems are hidden.

Problem Solving

General strategies (from Think Like a Programmer, V. A. Spraul)

- Start with what you know
  - Fully investigate a problem with the skills you have first.
  - Build confidence and momentum towards your goal.
  - You may learn more about the problem this way.
- Reduce the problem
  - Reduce scope by adding or removing constraints.
  - Work on a simpler problem that isn't easily divided.
  - Pinpoint where remaining difficulties lie.
- Look for analogies
  - Look for similarities to problems you've already solved.
  - Recognizing analogies improves speed and skill.
  - You need to build up a store of prior problems before you can find analogies.

Problem Solving

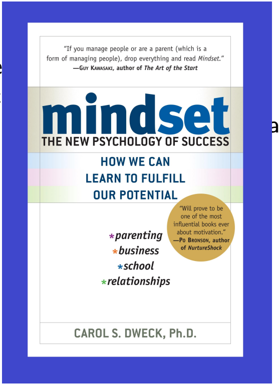
General strategies (from Think Like a Programmer, V. A. Spraul)

- Experiment
  - Try things and observe the results (this is not guessing!).
  - Trial-and-error is a valid approach to problem solving (not to be confused with guessing)
  - Make small test programs.
- Don't get frustrated
  - Everything will seem to take longer and be harder!
  - Avoiding frustration is a decision you make.
  - Go back to the plan, work on a different problem, or take a break.

Problem Solving

General strategies (from Think Like a Programmer, V. A. Spraul)

- Experiment
  - Try things and observe the results (this is not guessing!).
  - Trial-and-error is a valid approach to problem solving (not to be confused with guessing)
  - Make small test programs.
- Don't get frustrated
  - Everything will seem to take
  - Avoiding frustration is a dec
  - Go back to the plan, work o ake a break.


"If you manage people or are a parent (which is a form of managing people), drop everything and read Mindset."
—Guy Kawasaki, author of The Art of the Start
mindset
THE NEW PSYCHOLOGY OF SUCCESS
HOW WE CAN LEARN TO FULFILL OUR POTENTIAL
"Will prove to be one of the most influential books ever about motivation."
—Po Bronson, author of NurtureShock
*parenting
*business
*school
*relationships
CAROL S. DWECK, Ph.D.

Object

Oriented

Programming

software engineering

There is no one single best way of writing code

software engineering

There is no one single best way of writing code

- small program (HelloWorld)

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
    - main method
    - sequence of statements
- bigger programs

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
    - main method
    - sequence of statements
- bigger programs
    - reduce duplicate code with methods (functions)
    - collect data together (arrays)

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
    - main method
    - sequence of statements
- bigger programs
    - reduce duplicate code with methods (functions)
    - collect data together (arrays)
- modular programming
    - break up program into modules
    - each has independent functionality

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming
  - break up program into modules
  - each has independent functionality
- procedural programming

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
    - main method
    - sequence of statements
- bigger programs
    - reduce duplicate code with methods (functions)
    - collect data together (arrays)
- modular programming
    - break up program into modules
    - each has independent functionality
- procedural programming
    - a sequence of procedure calls that modify the state
    - Basic, Pascal, Fotran, C

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming
  - break up program into modules
  - each has independent functionality
- procedural programming
  - a sequence of procedure calls that modify the state
  - Basic, Pascal, Fotran, C, Java, C++

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming
  - break up program into modules
  - each has independent functionality
- procedural programming
  - a sequence of procedure calls that modify the state
  - Basic, Pascal, Fotran, C, Java, C++
- object oriented programming

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming
  - break up program into modules (classes in Java)
  - each has independent functionality
- procedural programming
  - a sequence of procedure calls that modify the state
  - Basic, Pascal, Fotran, C, Java, C++
- object oriented programming
  - a collection of interacting objects

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming
  - break up program into modules (classes in Java)
  - each has independent functionality
- procedural programming
  - a sequence of procedure calls that modify the state
  - Basic, Pascal, Fotran, C, Java, C++
- object oriented programming
  - a collection of interacting objects
  - objects have both `state` and `behaviour`

software engineering

- **large** projects are very **complex**

software engineering

- **large** projects are very **complex**
- humans are not perfect...

software engineering

- **large** projects are very **complex**
- humans are not perfect...
  - we make mistakes when solving complex problems

software engineering

- **large** projects are very **complex**
- humans are not perfect...
  - we make mistakes when solving complex problems
  - we make mistakes because we are lazy

software engineering

- **large** projects are very **complex**
- humans are not perfect...
  - we make mistakes when solving complex problems
  - we make mistakes because we are lazy
  - we cheat when we can

software engineering

- **large** projects are very **complex**

- humans are not perfect...
  - we make mistakes when solving complex problems
  - we make mistakes because we are lazy
  - we cheat when we can

- how can we reduce the impacts humans have on code?

software engineering

- **large** projects are very **complex**

- humans are not perfect...
  - we make mistakes when solving complex problems
  - we make mistakes because we are lazy
  - we cheat when we can

- how can we reduce the impacts humans have on code?
  - write less code

software engineering

- **large** projects are very **complex**
- humans are not perfect...
  - we make mistakes when solving complex problems
  - we make mistakes because we are lazy
  - we cheat when we can
- how can we reduce the impacts humans have on code?
  - write less code
  - prevent cheating

software engineering

- **large** projects are very **complex**
- humans are not perfect...
  - we make mistakes when solving complex problems
  - we make mistakes because we are lazy
  - we cheat when we can
- how can we reduce the impacts humans have on code?
  - write less code
  - prevent cheating
  - simplify the complexity (**abstraction**)

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
    - each procedure should do one thing (modularization)

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
    - each procedure should do one thing (modularization)

why might we want to program like this?

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
    - each procedure should do one thing (modularization)

why might we want to program like this?

- code is easier for us to read

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
    - each procedure should do one thing (modularization)

why might we want to program like this?

- code is easier for us to read
- code is easier to test/debug

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
  - each procedure should do one thing (modularization)

why might we want to program like this?

- code is easier for us to read
- code is easier to test/debug
- code is easier for others to read

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
  - each procedure should do one thing (modularization)

why might we want to program like this?

- code is easier for us to read
  - saves us time developing/maintaining/upgrading code
- code is easier to test/debug
  - saves us time developing/maintaining/upgrading code
- code is easier for others to read
  - saves us time developing/maintaining/upgrading code

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
  - each procedure should do one thing (modularization)

why might we want to program like this?

- code is easier for us to read
  - saves us time developing/maintaining/upgrading code
  - saves us money, time and our sanity
- code is easier to test/debug
  - saves us time developing/maintaining/upgrading code
  - saves us money, time and our sanity
- code is easier for others to read
  - saves us time developing/maintaining/upgrading code
  - saves us money, time and our sanity

object oriented programming (OOP)

a different approach to programming that focuses on objects interacting with each other (passing messages to each other)

three principles of object oriented programming:

- **encapsulation**
  - objects combine both data and operations on the data
  - objects have both `state` and `behaviour`
- **inheritance**
  - classes inherit data and operations from other classes
- **polymorphism**
  - objects can act like other objects. `dynamic binding` allows objects to determine which methods to use at runtime.

object oriented programming (OOP)

a different approach to programming that focuses on objects interacting with each other (passing messages to each other)

three principles of object oriented programming:

- **encapsulation**
  - objects combine both data and operations on the data
  - objects have both `state` and `behaviour`
  - allows us to model very complex real-world problems nicely

- **inheritance**
  - classes inherit data and operations from other classes
  - promotes code sharing and re-usability (write less code!)
  - [intuitive] hierarchical code organization

- **polymorphism**
  - objects can act like other objects. `dynamic binding` allows objects to determine which methods to use at runtime.
  - simplifies code understanding
  - standardizes method naming

object oriented programming (OOP)

a different approach to programming that focuses on objects interacting with each other (passing messages to each other)

three principles of object oriented programming:

- **encapsulation**
  - objects combine both data and operations on the data
  - objects have both `state` and `behaviour`
  - allows us to model very complex real-world problems nicely
- **inheritance**
  - classes inherit data and operations from other classes
  - promotes code sharing and re-usability (write less code!)
  - [intuitive] hierarchical code organization
- **polymorphism**
  - objects can act like other objects. `dynamic binding` allows objects to determine which methods to use at runtime.
  - simplifies code understanding
  - standardizes method naming
- OOP shines in BIG projects
  (don't be discouraged if it seems like a lot of work at first)

classes and objects

data type ▸ is a set of values and a set of operations defined on those values

classes and objects

data type ▸ is a set of values and a set of operations defined on those values

int ▸ the integers $-2,147,483,648 \rightarrow 2,147,483,647$
`+, -, *, /, %, <, >, <=, ...`

classes and objects

data type ▸ is a set of values and a set of operations defined on those values

int ▸ the integers $-2{,}147{,}483{,}648 \rightarrow 2{,}147{,}483{,}647$
`+, -, *, /, %, <, >, <=, ...`

String ▸ sequence of zero or more characters
+ (concatenation), `toUpper()`, etc

classes and objects

- data type ▸ is a set of values and a set of operations defined on those values

  - int ▸ the integers $-2,147,483,648 \rightarrow 2,147,483,647$
    `+, -, *, /, %, <, >, <=, ...`
  - String ▸ sequence of zero or more characters
    + (concatenation), `toUpper()`, etc

- Java has 8 primitive data types

classes and objects

data type
- is a set of values and a set of operations defined on those values

int
- the integers $-2,147,483,648 \rightarrow 2,147,483,647$
  `+, -, *, /, %, <, >, <=, ...`

String
- sequence of zero or more characters
  + (concatenation), `toUpper()`, etc

- Java has 8 primitive data types
- Many non-primitive data types are available (String, Date, etc)

classes and objects

data type
- is a set of values and a set of operations defined on those values

int
- the integers $-2,147,483,648 \rightarrow 2,147,483,647$
  `+, -, *, /, %, <, >, <=, ...`

String
- sequence of zero or more characters
  + (concatenation), `toUpper()`, etc

- Java has 8 primitive data types
- Many non-primitive data types are available (String, Date, etc)
- Java allows us to make our own data types
  - each `class` is a new data type
  - specifies data and operations on the data

classes and objects

data type
: is a set of values and a set of operations defined on those values

- a **class** is a data type (the cookie cutter)
  - specifies what data can be stored
    - instance attributes
  - defines operations on that data
    - instance methods

classes and objects

data type
: is a set of values and a set of operations defined on those values

- a **class** is a data type (the cookie cutter)
  - specifies what data can be stored
    - instance attributes
  - defines operations on that data
    - instance methods
  - can have its own data (cookie cutter model number, colour, etc)
    - class attributes / static attributes / class fields

classes and objects

data type ▸ is a set of values and a set of operations defined on those values

- a **class** is a data type (the cookie cutter)
  - specifies what data can be stored
    - instance attributes
  - defines operations on that data
    - instance methods
  - can have its own data (cookie cutter model number, colour, etc)
    - class attributes / static attributes / class fields
  - can have its own functionality (bottle opener?)
    - class methods / static methods (functions)

classes and objects

data type ▸ is a set of values and a set of operations defined on those values

- a **class** is a data type (the cookie cutter)
  - specifies what data can be stored
    - instance attributes
  - defines operations on that data
    - instance methods
  - can have its own data (cookie cutter model number, colour, etc)
    - class attributes / static attributes / class fields
  - can have its own functionality (bottle opener?)
    - class methods / static methods (functions)

- an **object** is an instantiation of a class (the cookie)
  - holds data (the **state** of the object)
    - instance attributes
  - has operations built-in to it (the **bahaviour** of the object)
    - instance methods

anatomy of a class (part I)

```java
public class MyClass{
    /* instance attributes */
    public int a;
    private String s;
    ...

    /* constructors */
    public MyClass(){ ... }
    public MyClass(int x){ ... }
    ...

    /* instance methods */
    public int addOne(){...}
    ...
    ...
    /* class methods */
    public static void main(String[] args){...}
}
```

anatomy of a class (part I)

```
public class MyClass{
    /* instance attributes */          ← this will define the STATE
    public int a;                                       (set of values)
    private String s;
    ...

    /* constructors */                            ← initialization code
    public MyClass(){ ... }
    public MyClass(int x){ ... }
    ...

    /* instance methods */          ← this defines the BEHAVIOUR
    public int addOne(){...}               (operations on the values)
    ...
    ...
    /* class methods */                                ← class methods
    public static void main(String[] args){...}
}
```

basic class

```java
public class Student{
    /* instance attributes */
    public String name;
    public int id;
}
```

basic class

```
public class Student{
    /* instance attributes */
    public String name;
    public int id;
}
```

- simple aggregation of data
- array was good for collecting together data of the same type (doesn't work for different data though)
- this is essentially a `record` in Pascal or a `struct` in C

basic class

```
public class Student{
    /* instance attributes */
    public String name;
    public int id;
}
```

- declare variable of type `Student` like any other variable
  - `Student s;`
- instantiate the actual object with `new` and constructor
  - `s = new Student();`
- access attributes with `dot` operator
  - `s.name = "fig";`

basic class

```
Student s = new Student();
```

- `Student()` is a constructor for Student class
- Java will provide a default constructor
  - if and only if you do not provide any constructors
  - default constructor has no input parameters
- you can define as many constructors as you see fit
  - Java allows method `overloading`
  - Java methods uniquely specified by name and input arguments
  - you can have many methods with the same name

basic class

```
public class Student{
    public String name;
    public int id;

    public Student(){
        name = "none";
        id = -1;
    }
}
```

basic class

```
public class Student{
    public String name;
    public int id;

    public Student(){
        name = "none";
        id = -1;
    }
}
```

- constructor has no return value
  (this is different than returning nothing; void)

basic class

```java
public class Student{
    public String name;
    public int id;

    public Student(){
        name = "none";
        id = -1;
    }
}
```

- constructor has no return value
  (this is different than returning nothing; void)
- constructor name is identical to class name

basic class

```
public class Student{
    public String name;
    public int id;

    public Student(){
        name = "none";
        id = -1;
    }
}
```

- constructor has no return value
  (this is different than returning nothing; void)
- constructor name is identical to class name
- executes initialization/start-up code when we create an object

basic class

```
public class Student{
    public String name;
    public int id;

    public Student(){
        name = "none";
        id = -1;
    }
}
```

- constructor has no return value
  (this is different than returning nothing; void)
- constructor name is identical to class name
- executes initialization/start-up code when we create an object
- constructors are not methods

basic class

```
public class Student{
    public String name;
    public int id;

    public Student(String name, int id){
             name = name;
             id = id;
    }
}
```

basic class

```
public class Student{
    public String name;
    public int id;

    public Student(String name, int id){
        this.name = name;
        this.id = id;
    }
}
```

- Java keyword `this`
  - used in constructors and instance methods
  - a reference to the current object
  - has other uses we'll discuss later
- `this` is needed here because attributes `name` and `id` are not in scope (the input parameters `name` and `id` are in scope)

basic class

```
public class Student{
    public String name;
    public int id;

    public Student(String nameInit, int idInit){
            name = nameInit;
            id = idInit;
    }
}
```

- `this` is not needed here

basic class

```
public class Student{
    public String name;
    public int id;

    public Student(String nameInit, int idInit){
        this.name = nameInit;
        this.id = idInit;
    }
}
```

- this is not needed here
- you can still use it though!
- I will often include it in constructors for this course

basic class

```java
public class Student{
    public String name;
    public int id;

    public Student(String nameInit, int idInit){
        this.name = nameInit;
        this.id = idInit;
    }

    public Student(String nameInit){
        this.name = nameInit;
        this.id = -1;
    }
}
```

- Java allows method and constructor **overloading**
- can have as many constructors as is useful

basic class

```
public class Student{
    public String name;
    public int id;



}
```

- class does not need to have a constructor specified

basic class

```
public class Student{
    public String name;
    public int id;

    public Student(){
    }
}
```

- class does not need to have a constructor specified
- Java automatically provides a zero argument default constructor if none are specified
    - it does nothing

basic class

```
public class Student{
    public String name;
    public int id;

    public Student(){
    }
}
```

- class does not need to have a constructor specified
- Java automatically provides a zero argument default constructor if none are specified
  - it does nothing
- Java does this only if NO constructors are specified

let’s take a break...
for 5 minutes

encapsulation (again)

encapsulation can also refer to a language mechanism for restricting access to some of the object's components. [wiki]

- often called **information hiding**
- related to idea of **separation of concerns**
    - actual code and how you use the code are independent
- access to data is restricted
    - through **getter** and **setter** methods (if accessible) (this is the interface in which you access data)
    - some data is completely hidden within the object
- why would we want to do this?

encapsulation (again)

encapsulation can also refer to a language mechanism for restricting access to some of the object's components. [wiki]

- often called **information hiding**
- related to idea of **separation of concerns**
  - actual code and how you use the code are independent
- access to data is restricted
  - through **getter** and **setter** methods (if accessible)
    (this is the interface in which you access data)
  - some data is completely hidden within the object
- why would we want to do this?
  - what happens if you change how you store your data?
  - if someone has access to a variable, will they modify it?

encapsulation

```
public class Student{
    public String name;
    public int id;

    public Student(String name, int id)
        {this.name = name; this.id = id; }
}
```

encapsulation

```
public class Student{
    private String name;
    private int id;









    public Student(String name, int id)
        {this.name = name; this.id = id; }
}
```

encapsulation

```
public class Student{
    private String name;
    private int id;

    public String getName()
        {return this.name;}

    public void setName(String newName)
        // setter for name
        {this.name = newName;}

    public Student(String name, int id)
        {this.name = name; this.id = id; }
}
```

encapsulation

```
public class Student{
    private String name;
    private int id;

    public String getName()
        {return this.name;}

    public String setName(String newName)
        // setter for name
        {this.name = newName; return this.name;}

    public Student(String name, int id)
        {this.name = name; this.id = id; }
}
```

encapsulation

```
public class Student{
    private String name;
    private int id;

    public String getName()
        {return this.name;}

    public Student setName(String newName)
        // setter for name
        {this.name = newName; return this;}

    public Student(String name, int id)
        {this.name = name; this.id = id; }
}
```

encapsulation

```
public class Student{
    private String name;
    private int id;

    public String getName()
        {return this.name;}

    public boolean setName(String newName, Cred cred){
        // setter for name
        if( isValidCredential(cred)){
            this.name = newName;
            return true;
        }else{
            return false;
        }
    }

    public Student(String name, int id)
        {this.name = name; this.id = id; }
}
```

Java's Object class

```
public class Object{
    /* no attributes */

    /* single constructor */
    public Object(){}

    /* 11 methods */
    public String toString(){...}
    public int hashCode(){...}
    public boolean equals(Object obj){...}
    ...
}
```

- `java.lang.Object`
- this is Java's basic non-primitive type

Java's Object class

```
public class Student {
    /* attributes */
    public String name;
    public int id;
}
```

Java's Object class

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
}
```

- implicit inheritance to `Object` if none given
- Java keyword `extends` used for inheritance

Java's Object class

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
}
```

- implicit inheritance to `Object` if none given
- Java keyword `extends` used for inheritance
- when we inherit from a class
  - we get all public attributes from the parent class
  - we get all public methods from the parent class
  - we get none of the constructors

Java's Object class

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
}
```

- implicit inheritance to `Object` if none given
- Java keyword `extends` used for inheritance
- when we inherit from a class
  - we get all public attributes from the parent class
  - we get all public methods from the parent class
  - we get none of the constructors
- we say that Student is an Object
  - this is the "is-a" relationship

Java's Object class

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
}
```

- implicit inheritance to `Object` if none given
- Java keyword `extends` used for inheritance
- when we inherit from a class
  - we get all public attributes from the parent class
  - we get all public methods from the parent class
  - we get none of the constructors
- we say that Student is an Object
  - this is the "is-a" relationship
- we say that Student has a String
  - this is the "has-a" relationship
  - this is class composition (not inheritance)

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
}
```

- a class can only have one parent class
- every class, except `Object`, has exactly one parent class
- we get a hierarchy of classes
    - a family tree of classes

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
}
```

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
}
```

- we say that
  - Student is a child class of Object
  - Student is a subclass of Object
  - Student is a derived class of Object
  - Student is a descendent of Object

inheritance

```java
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
}
```

- we say that
  - Student is a child class of Object
  - Student is a subclass of Object
  - Student is a derived class of Object
  - Student is a descendent of Object

- we say that
  - Object is a parent class of Student
  - Object is a super class of Student
  - Object is a ancestor of Student

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
}
```

- what do we get from `Object`?

inheritance - method overriding

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    public String toString(){...}
}
```

inheritance - method overriding

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    public String toString(){...}
}
```

- method overriding
  - allows us to redefine a parent's (or grandparent's) method definition

inheritance - method overriding

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    public String toString(){...}
}
```

- method `overriding`
  - allows us to redefine a parent's (or grandparent's) method definition
- which method is executed?

inheritance - method overriding

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    public String toString(){...}
}
```

- method `overriding`
  - allows us to redefine a parent's (or grandparent's) method definition
- which method is executed?
  - Java first looks in current class
  - if method is not defined, look at parent class
  - if method is not defined, look at parent class
  - ...
  - get method from `Object`

inheritance - method overriding

let's see some examples...