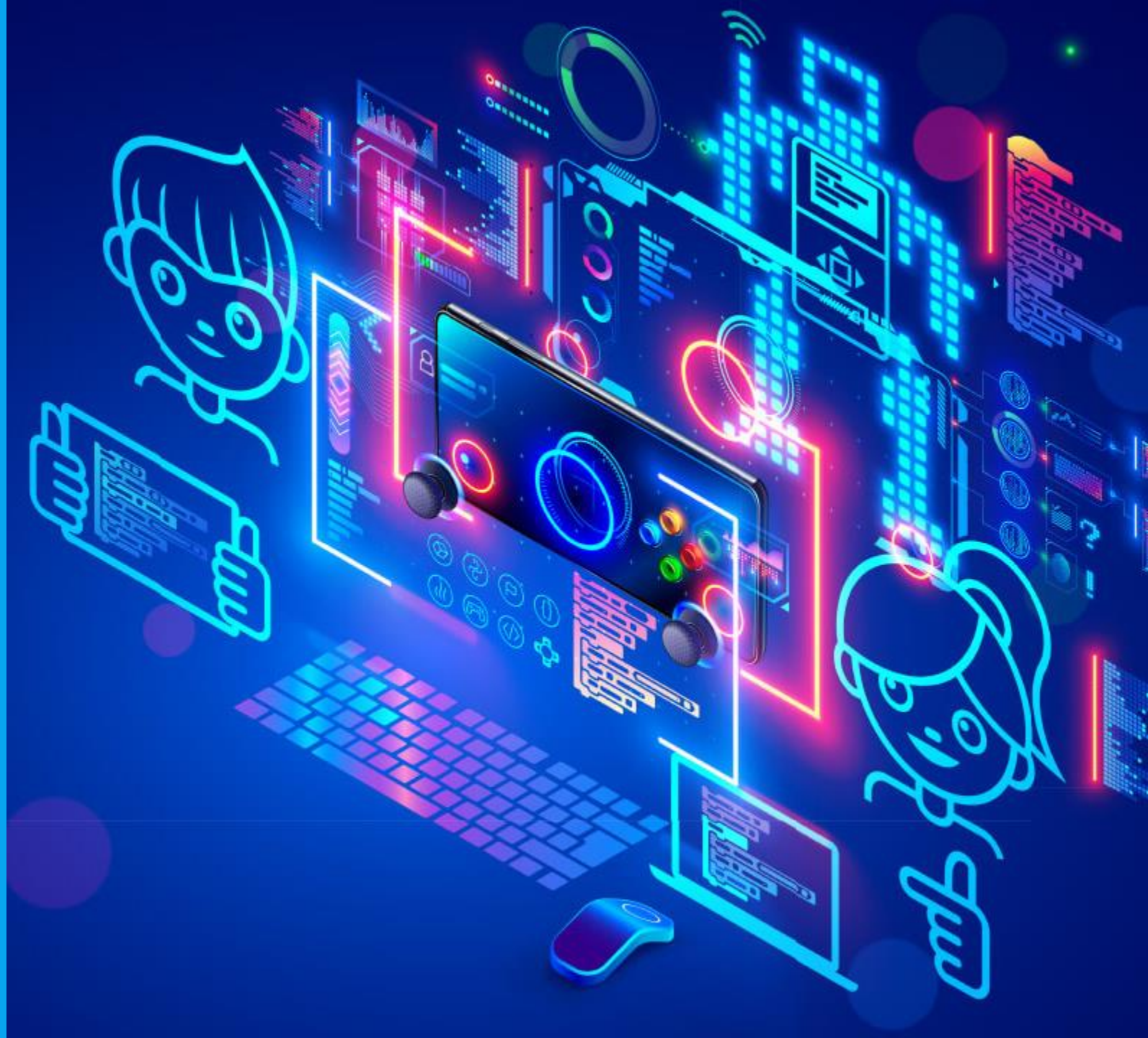# CS 24 AP Computer Science A Review

Week 5: Program Design and Analysis and 1-D Array
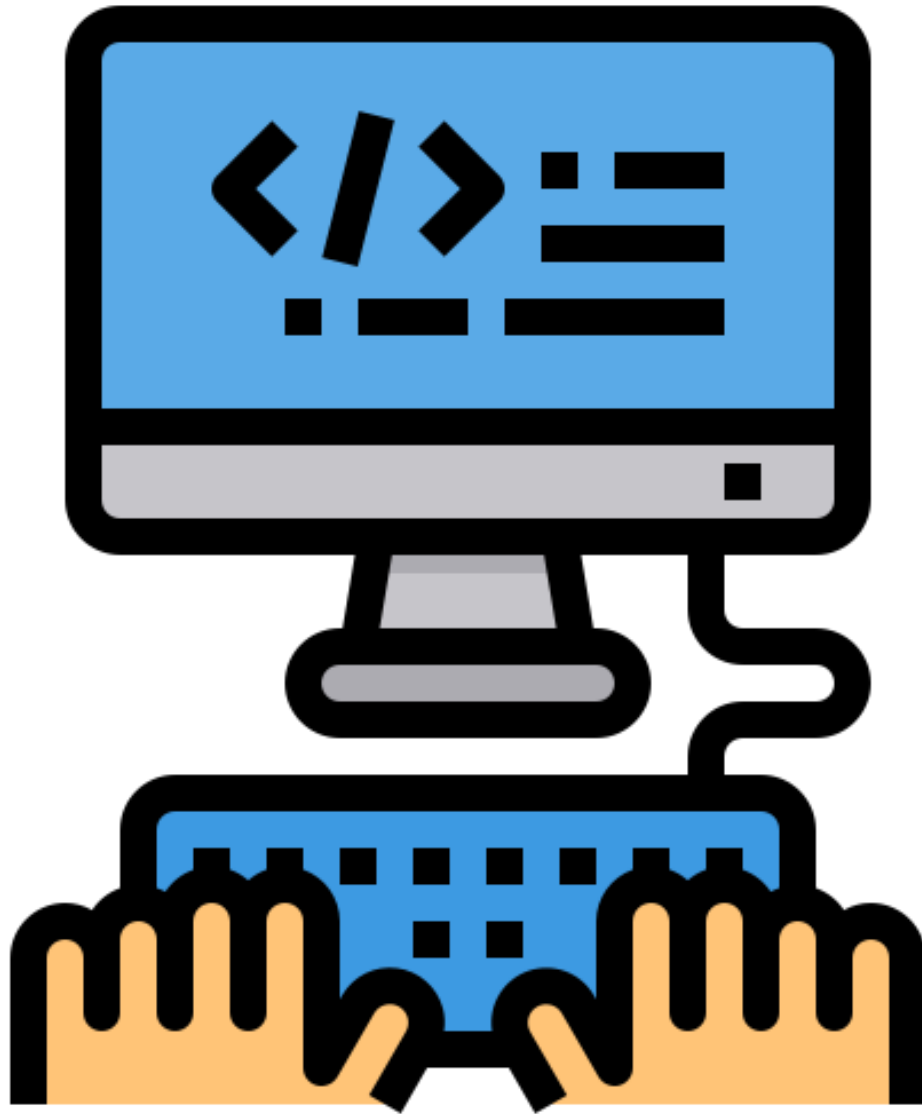
DR. ERIC CHOU
IEEE SENIOR MEMBER

Topics

- Software Development

- Object-Oriented Design

- Is-A Relationship and Has-A Relationship

- Unified Modeling Language

- Implementation (Coding)

- Program Analysis

- One Dimensional Array (Quiz Next Week)
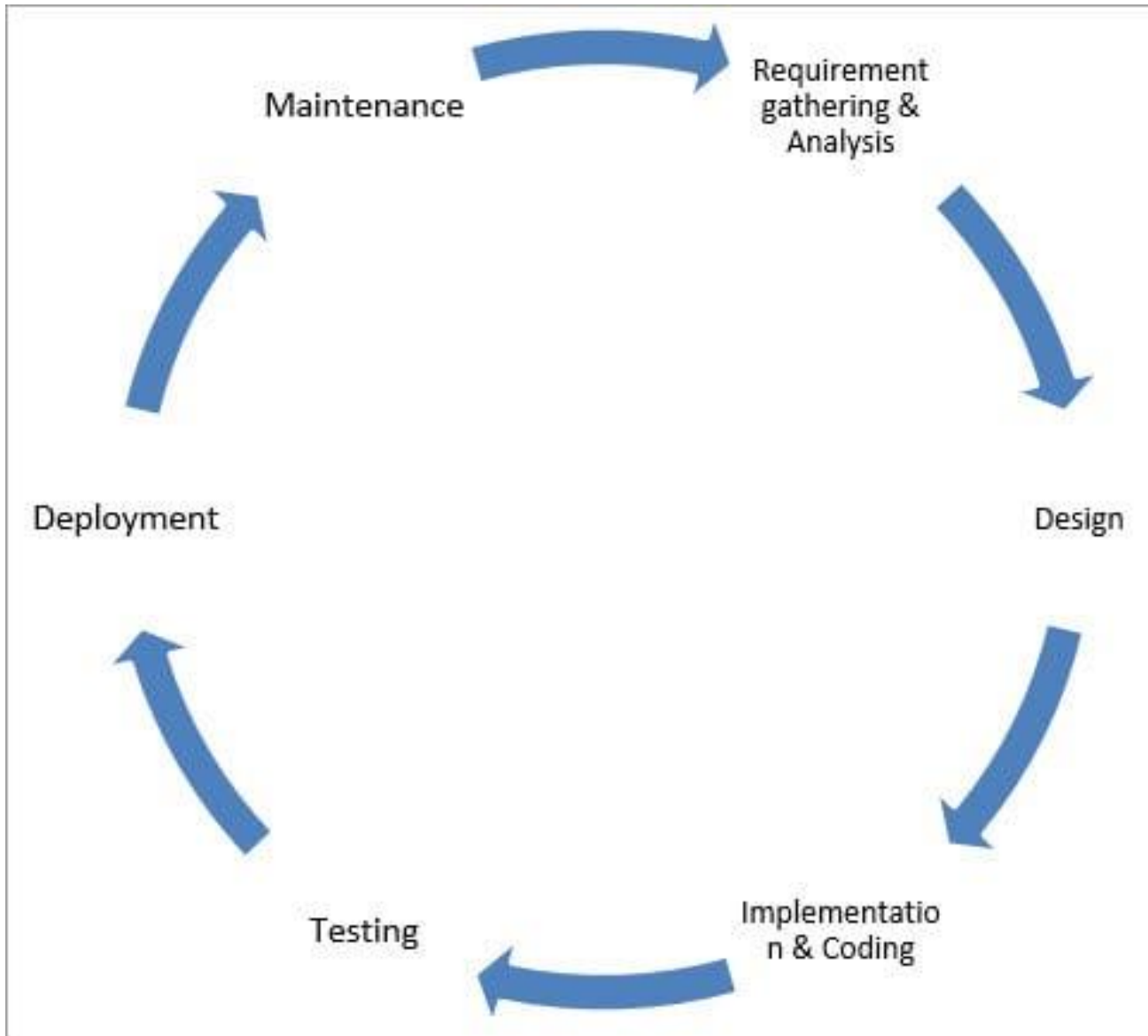
- Array Processing (Quiz Next Week)

Software Development Life-Cycle (SDLC)

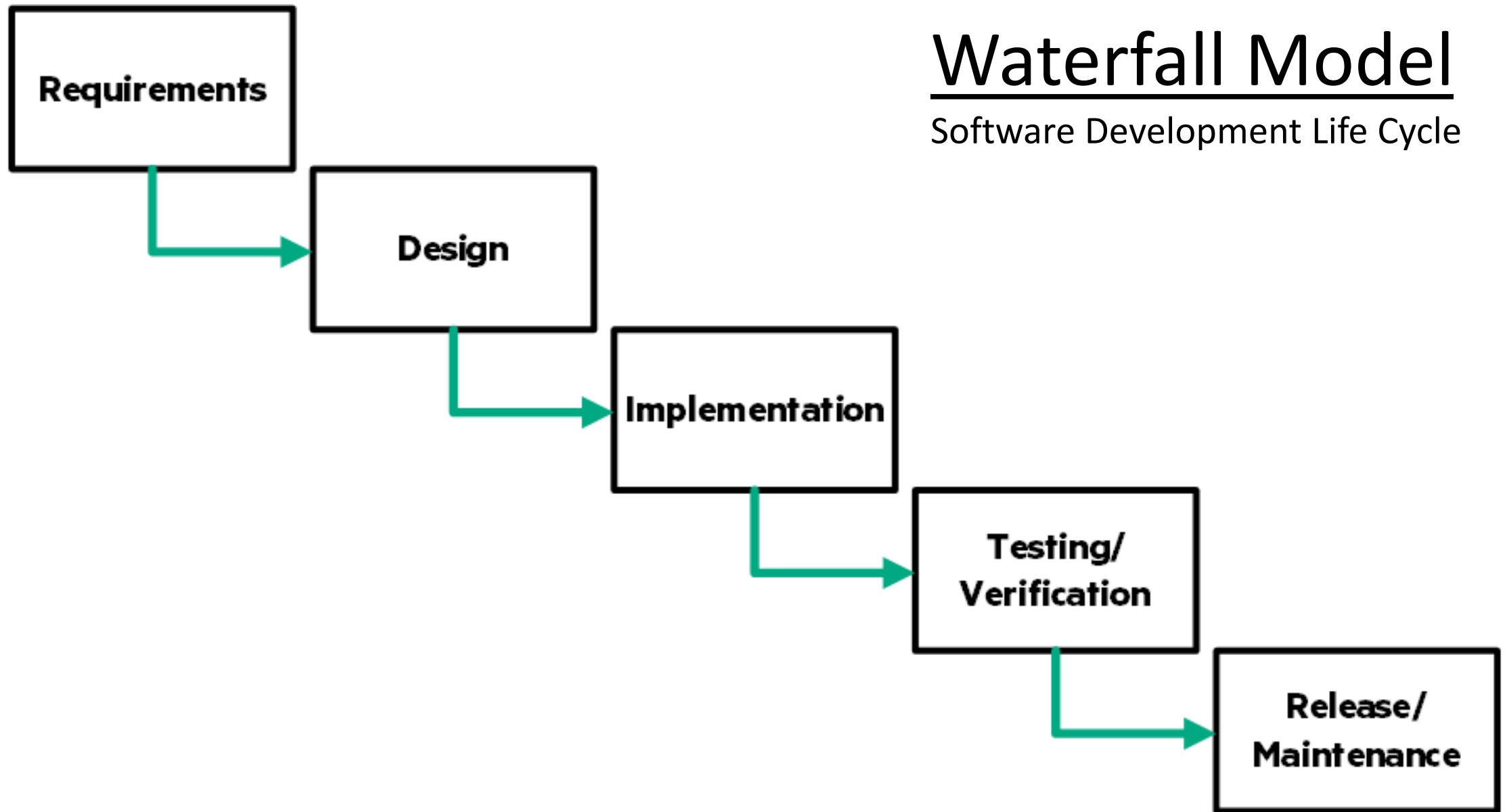Section 1

Software Engineering

- Software Engineering is the study of designing, developing, and maintaining software.

- Over the years, many different software development models have been designed that help developers create quality software.

Software Design Life Cycle

Software Development Models

- **Prototyping** – an approximation of a final system is built, tested, and reworked until it is acceptable. The complete system is then developed from this prototype.

- **Incremental Development** – The software is designed, implemented, and tested a little bit at a time until the product is finished.

- **Rapid Application Development** – the user is actively involved in the evaluation of the product and modifications are made immediately as problems are found. Radical changes in the system are likely at any moment in the process.

Software Development Models

- **Agile Software Development** – the developers offer frequent releases of the software to the customer and new requirements are generated by the users. Short development cycles are common in this type of development.

- **Waterfall Model** – The progress of development is sequential and flows downward like a waterfall. Steps include conception, initiation, analysis, design, construction, testing, implementation, and maintenance.

Program Specification

Section 2

Program Specifications

- A program specification describes the results that a program is expected to produce -- its primary purpose is to be understood not executed. Specifications provide the foundation for programming methodology.

- A specification is a **technical contract** between a programmer and his/her client and is intended to provide them with a mutual understanding of a program.

Program Specifications

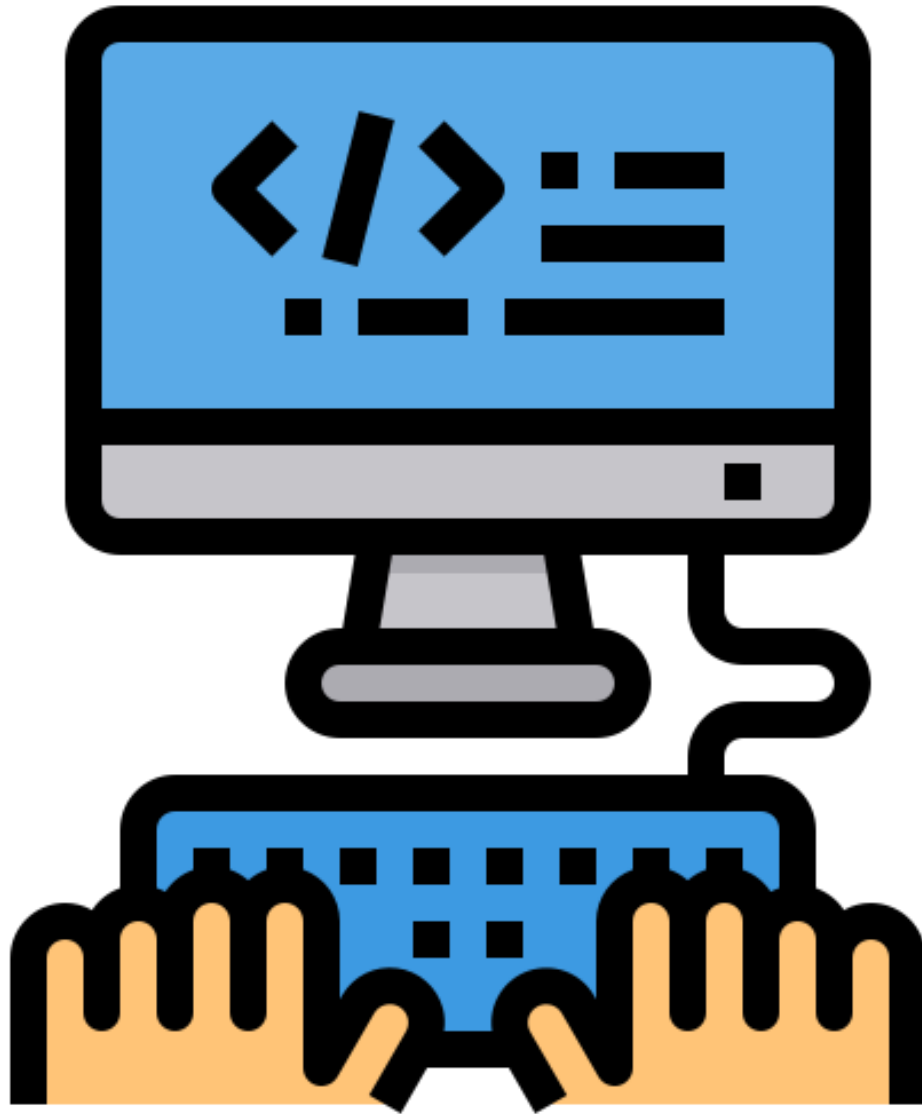- A client uses the specification to guide his/her use of the program; a programmer uses the specification to guide his/her construction of the program. A complex specification may engender sub specifications, each describing a sub-component of the program.

- The construction of these sub-components may then be delegated to other programmers so that a programmer at one level becomes also a client at another.

Program Specification

- The specification is an explicit written description of the project. Typically, it is based on a customer's (or a teacher's!) requirements. The first step in writing a program is to analyze the specification. Make sure you understand it, and clarify with the customer anything that is unclear.

System Design

Section 3

Program Design

- Even for a small-scale program a good design can save programming time and enhance the reliability of the final program. The design is a fairly detailed plan for solving the problem outlined in the specification. It should include all objects that will be used in the solution, the data structures that will implement them, plus a detailed list of the tasks to be performed by the program.

- A good design provides a fairly detailed overall plan at a glance, without including the minutiae of Java code.

Design Class Hierarchy

- Top-down and bottom-up are two ways of approaching class hierarchy design. Top-down is also referred to as functional decomposition. The two designs are different only in their approach to the problem.
- Top-down design starts with the big picture, whereas a bottom-up design starts with the details. They each work toward the other, but where they begin is different.

Top Down and Bottom Up Design

Vision → **Design** → **Develop**

VS

Reuse → **Modularize** → **Solve**
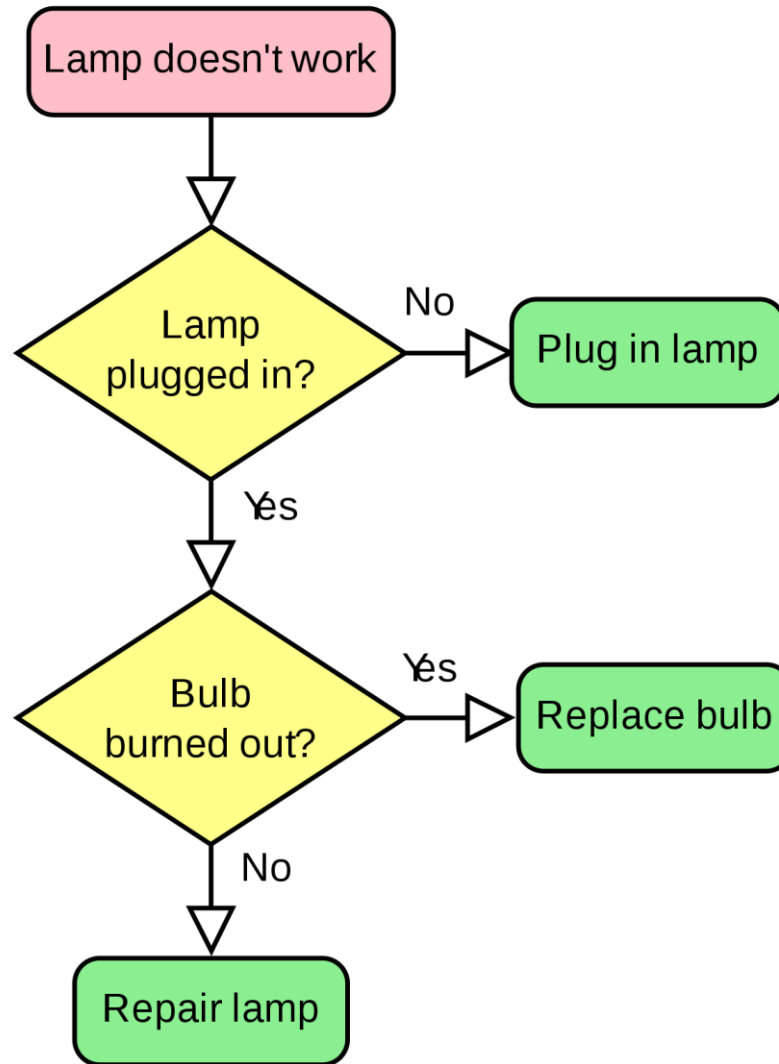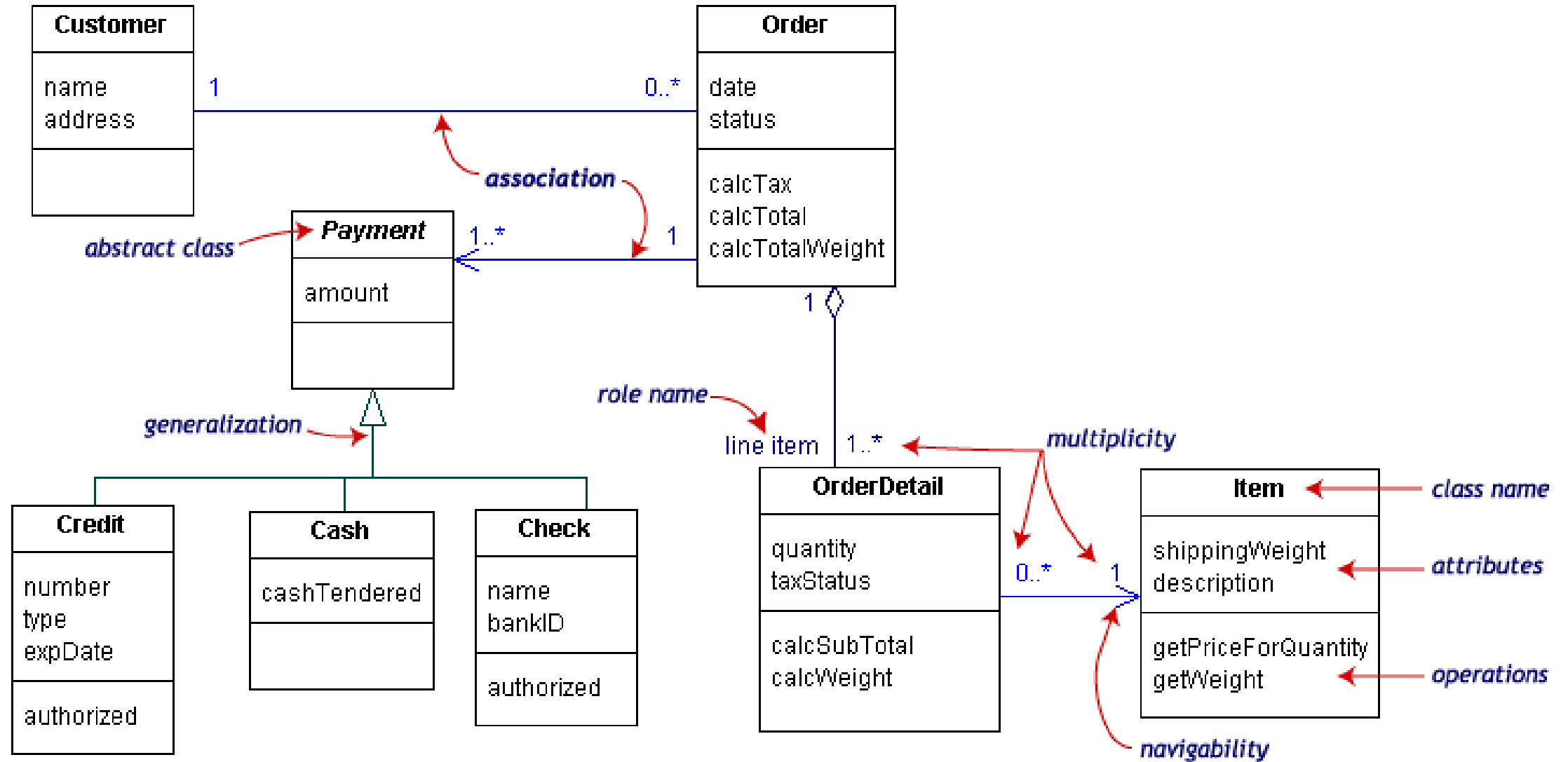
Pseudocode to Calculate the Sum & Average fo 10 Numbers

```
begin
        initialize counter to 0
        initialize accumulator to 0
        loop
                read input from keyboard
                accumulate input
                increment counter
        while counter < 10
        calculate average
        print sum
        print average
end
```

Design Methodology

Section 3

Procedural Abstraction

- Procedural Abstractions organize instructions.

$\downarrow$

> **Function Power**
> **Give me two numbers (base & exponent)**
> **I'll return base^{exponent}**
>
> $\downarrow$
>
> **Implementation**

Data Abstraction

User Program

private

name: string
id: int

public

get_name()
set_name(str)

public

get_id()
set_id(int)
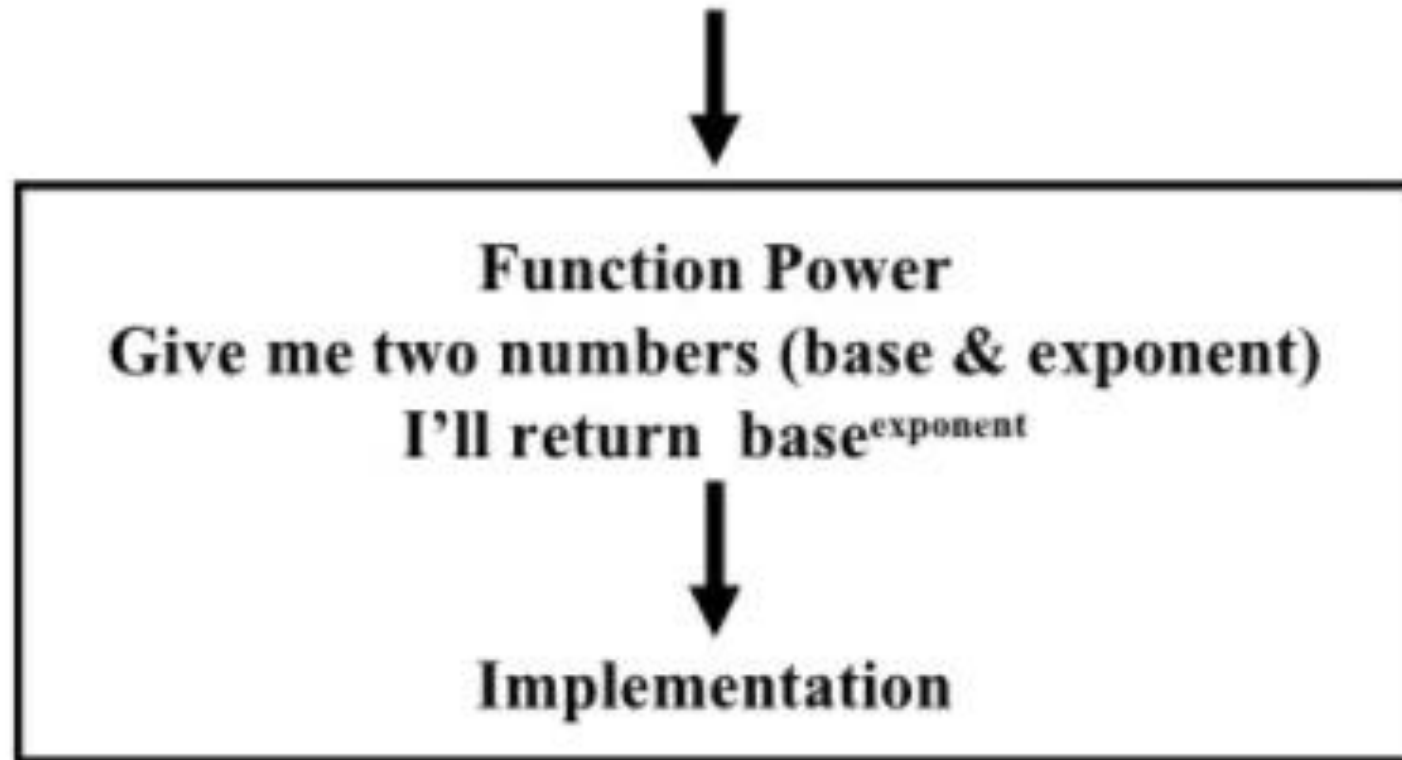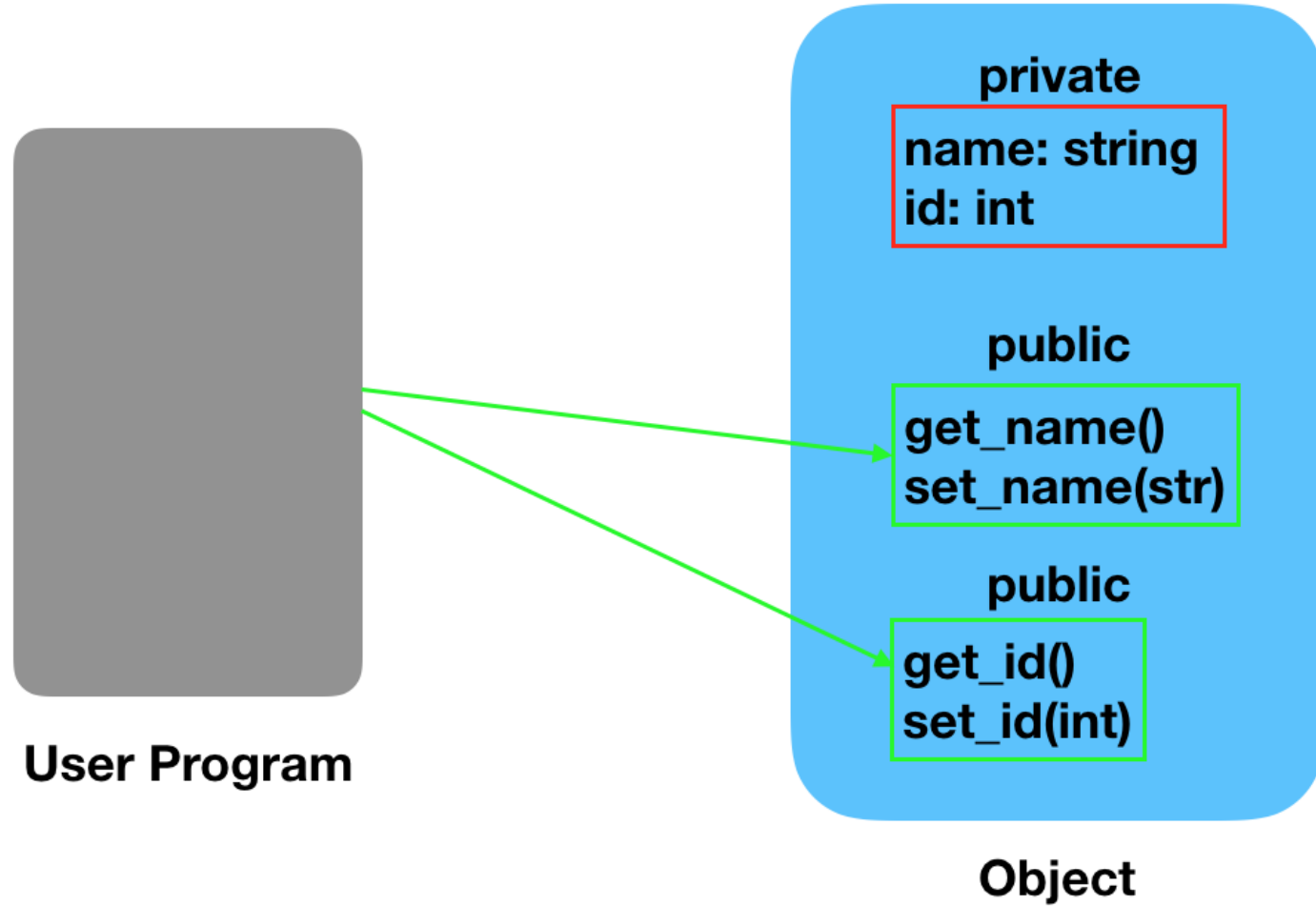
Object

Object-Oriented Programming
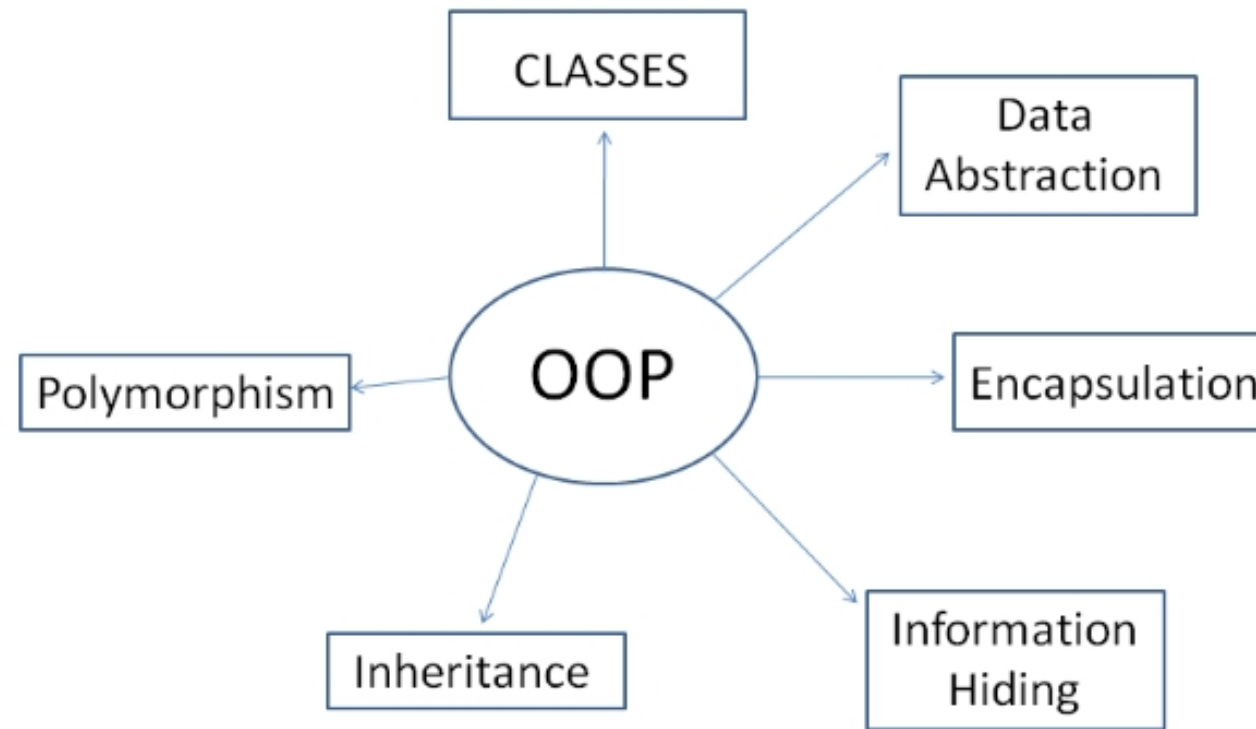
Classes

Objects (or Instances)

Vehicle

GasVehicle

ElectricVehicle

Car

Motorcycle

ElectricCar

genericVehicle15

myCar

myMotorcycle

car37

carForJim
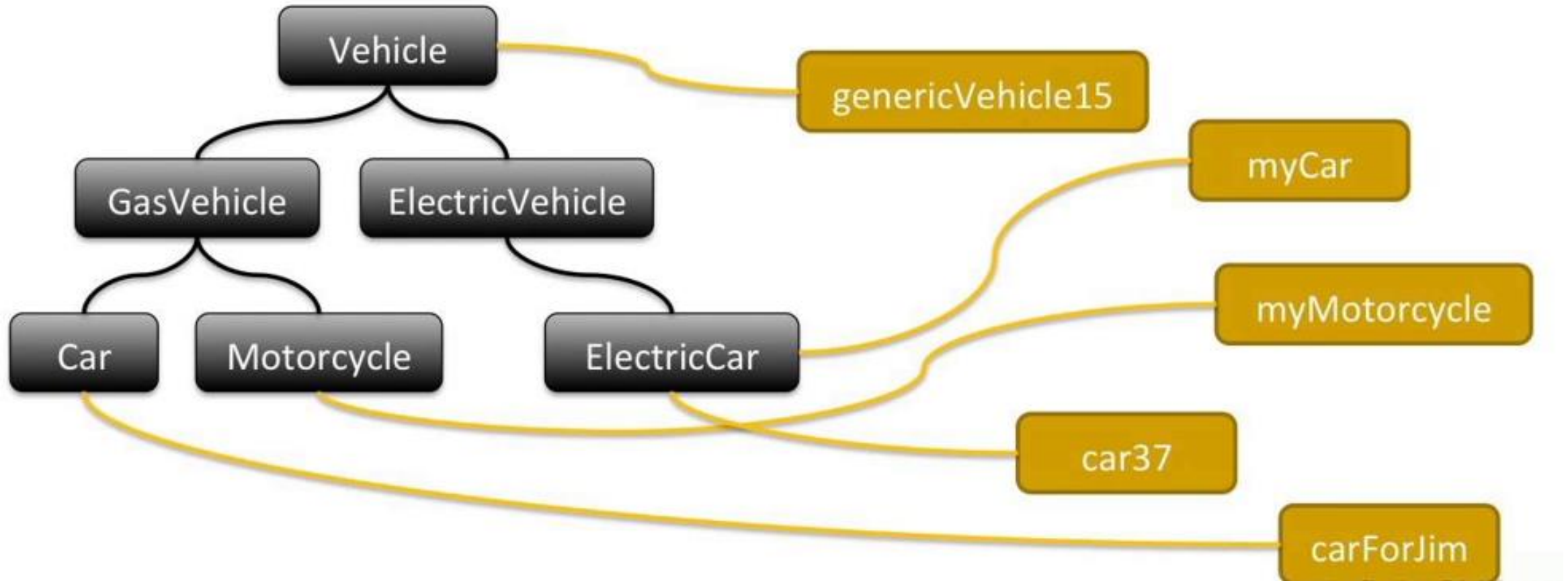
Object-Oriented Design

Section 2

Object-Oriented Programming

Object-oriented programming has been the dominant programming methodology since the mid 1990s.

Here are the steps in object-oriented design:
- Identify classes to be written.
- Identify behaviors (i.e., methods) for each class.
- Determine the relationships between classes.
- Write the public method headers for each class.
- Implement the methods.

Identifying Classes

- Identify the objects in the program by picking out the nouns in the program specification. Ignore pronouns and nouns that refer to the user. Select those nouns that seem suitable as classes, the "big-picture" nouns that describe the major objects in the application. Some of the other nouns may end up as attributes of the classes.

- Many applications have similar object types: a low-level basic component; a collection of low-level components; a controlling object that puts everything together; and a display objectthat could be a GUI (graphical user interface) but doesn't have to be.

Example 1

- Write a program that maintains an inventory of stock items for a small store.

Nouns to consider: inventory, item, store.

Basic Object: StockItem

Collection: Inventory (a list of StockItemS)

Controller: Store (has an Inventory, uses a StoreDisplay)

Display: StoreDisplay (could be a GUI)

MVC Architecture Pattern
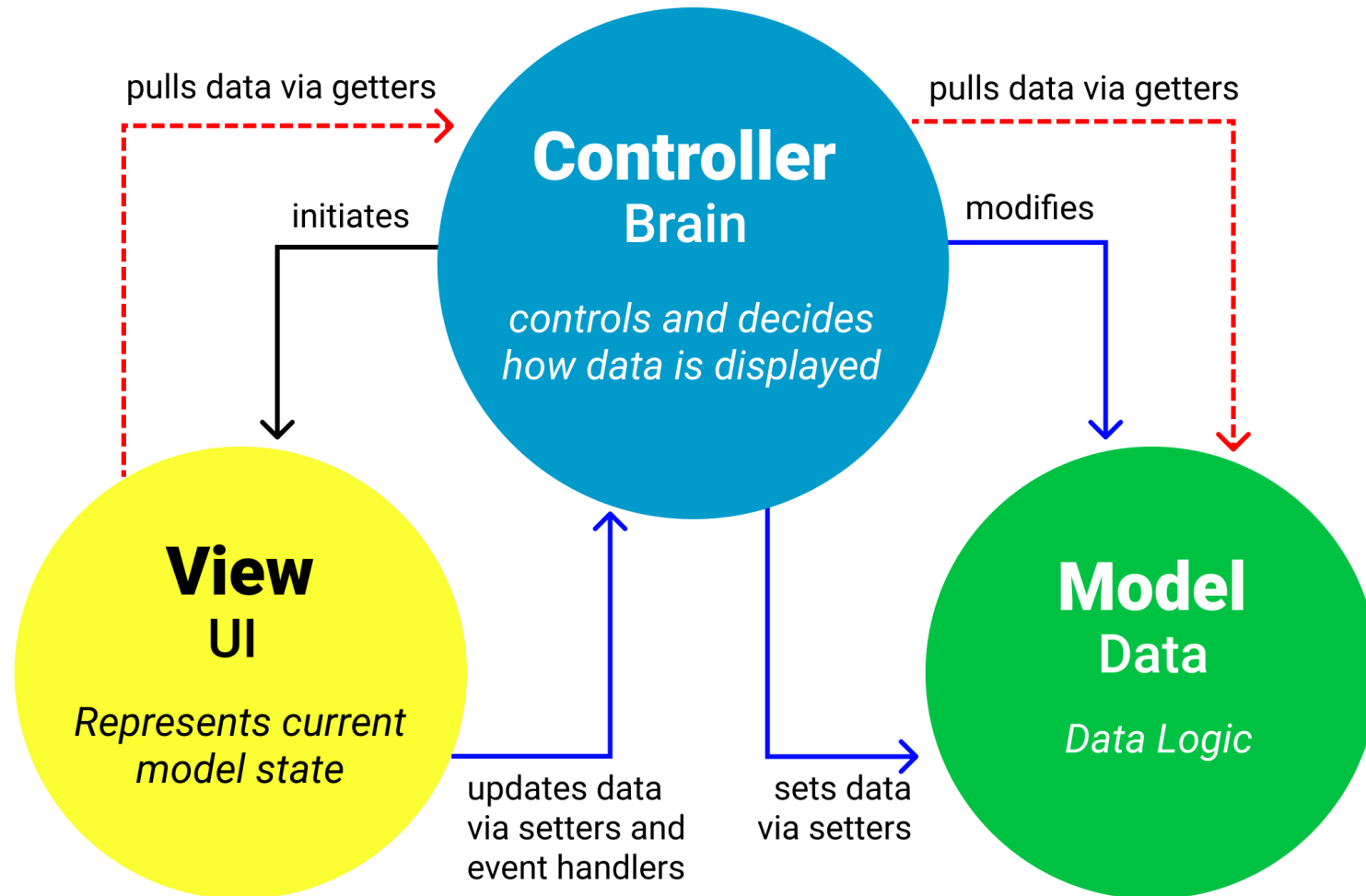
Example 2

- Write a program that simulates a game of bingo. There should be at least two players, each of whom has a bingo card, and a caller who calls the numbers.

    Nouns to consider: game, players, bingo card, caller.

    Basic Objects: BingoCard, Caller

    Collection: Players (each has a BingoCard)

    Controller: GameMaster (sets up the Players and Caller)

    Display: BingoDisplay (shows each player's card and displays winners, etc.)

Example 3

- Write a program that creates random bridge deals and displays them in a specified format. (The specification defines a "deal" as consisting of four hands. It also describes a deck of cards, and shows how each card should be displayed.)

    Nouns to consider: deal, hand, format, deck, card.

    Basic Object: Card

    Collection:

    - Deck (has an array of Cards)
    - Hand (has an array of Cards)
    - Deal (has an array of Hands)
    - Dealer (has a Deck, or several Decks)

    Controller: Formatter (has a Deal and a TableDisplay)

    Display: TableDisplay (could be a GUI)

Identifying Behaviors

- Find all verbs in the program description that help lead to the solution of the programming task. These are likely behaviors that will probably become the methods of the classes. Now decide which methods belong in which classes.

- Recall that the process of bundling methods and data fields into a class to enable its data to be hidden is called data encapsulation.

Identifying Behaviors

- Think carefully about who should do what. Do not ask a basic object to perform operations for the group. For example, a StockItem should keep track of its own details (price, description, how many on the shelf, etc.) but should not be required to search for another item. A Card should know its value and suit but should not be responsible for keeping track of how many cards are left in a deck.

- A Caller in a bingo game should be responsible for keeping track of the numbers called so far and for producing the next number but not for checking whether a player has bingo: That is the job of an individual player (element of Players) and his BingoCard.

Identifying Behaviors

- You will also need to decide which data fields each class will need and which data structures should store them. For example, if an object represents a list of items, consider an array or `ArrayList` as the data structure.

Is-A Relationship and Has-A Relationship

Section 3

Determining Relationships Between Classes

INHERITANCE RELATIONSHIPS

- Look for classes with common behaviors. This will help identify inheritance relationships. Recall the **is-a** relationship-if object 1 **is-a** object2, then object2 is a candidate for a superclass.

Determining Relationships Between Classes

COMPOSITION RELATIONSHIPS
- Composition relationships are defined by the **has-a** relationship. For example, a Nurse has-a Uniform. Typically, if two classes have a composition relationship, one of them contains an instance variable whose type is the other class.
- Note that a wrapper class always implements a has-a relationship with any objects that it wraps.

UML Diagrams

- An excellent way to keep track of the relationships between classes and show the inheritance hierarchy in your programs is with a **UML** (Unified Modeling Language) diagram. This is a standard graphical scheme used by object-oriented programmers. Although it is not part of the AP subset, on the AP exam you may be expected to interpret simple **UML** diagrams and inheritance hierarchies.

- Here is a Simplified version of the **UML** rules:
  - Represent classes with rectangles.
  - Show the is-a relationship between classes with an open up-arrow.
  - Show the has-a relationship with a down arrow or sideways arrow (indicates composition) .

Example

- From this diagram you can see at a glance that GoodPlayer and BadPlayer are subclasses of a class Player, and that every Player has a Board and a ScoreCard, while only the BadPlayer has a Tutor.
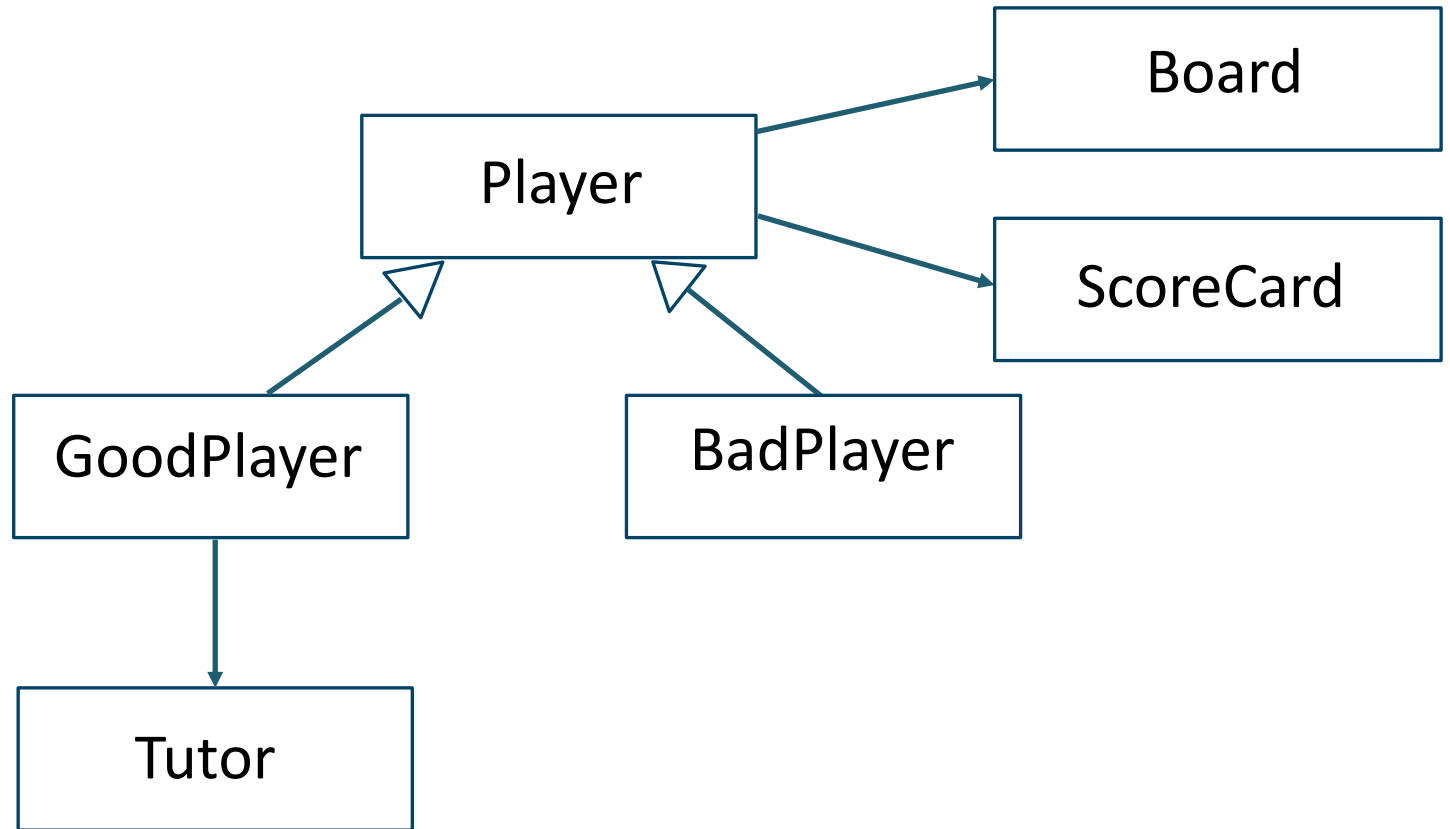
Implementing Classes

- For each method in a class, list all of the other classes needed to implement that particular method. These classes are called collaborators. A class that has no collaborators is independent.

Implementing Classes

- To implement the classes, often an incremental, bottom-up approach is used. This means that independent classes are fully implemented and tested before being incorporated into the overall project.

- Typically, these are the basic objects of the program, like StockItem, Card, and BingoCard. Unrelated classes in a programming project can be implemented by different programmers.

Implementing Classes
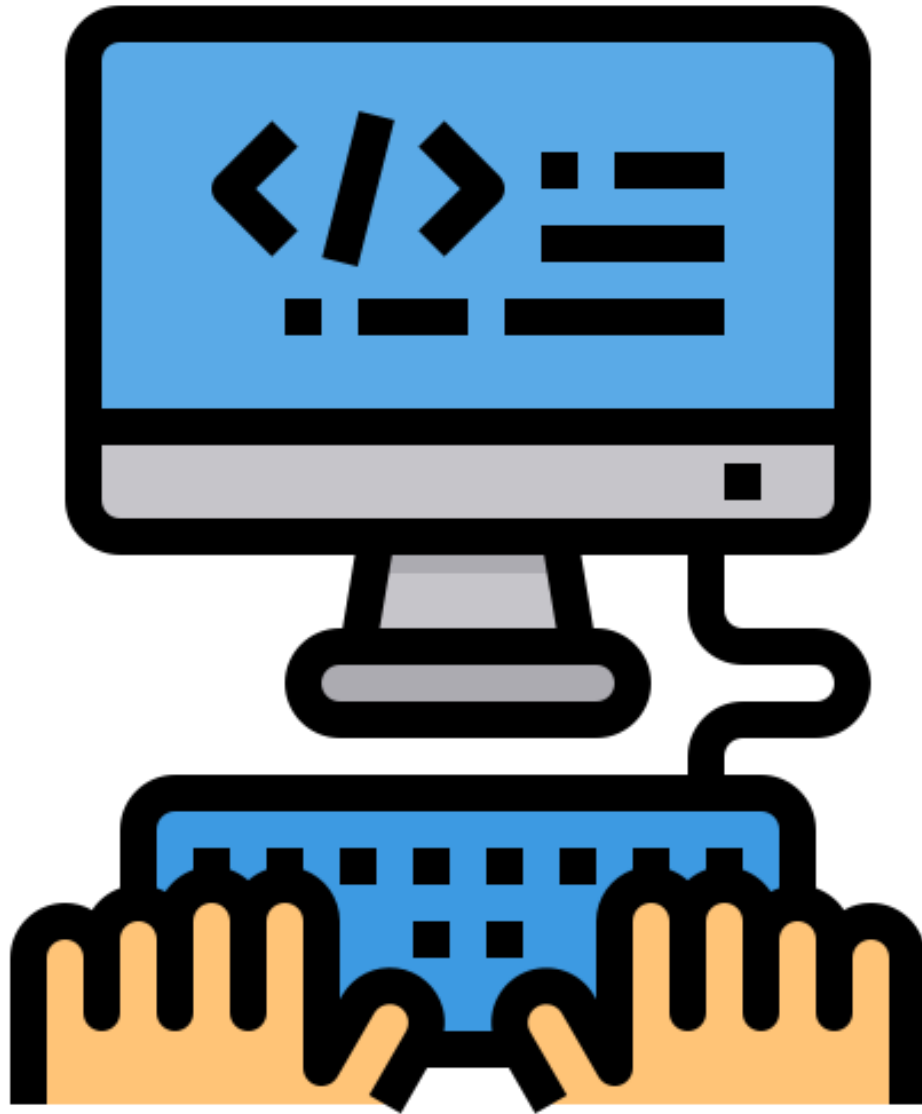
- Note that a class can be tested using a dummy Tester class that will be discarded when the methods of the class are working. Constructors, then methods, should be added, and tested, one at a time. A driver class that contains a main method can be used to test the program as you go.

- The purpose of the driver is to test the class fully before incorporating it as an object in a new class.

Implementing Classes

- When each of the independent classes is working, classes that depend on just one other class are implemented and tested, and so on. This may lead to a working, bare bones version of the project. New features and enhancements can be added later.

- Design flaws can be corrected at each stage of development. Remember, a design is never set in stone: It simply guides the implementation.

Implementation (Coding)

Section 5

Top-Down Development

- In a top-down design, the programmer starts with an overview of the program, selecting the highest-level controlling object and the tasks needed. During development of the program, subsidiary classes may be added to simplify existing classes.

Implementing Methods

PROCEDURAL ABSTRACTION

- A good programmer avoids chunks of repeated code wherever possible. To this end, if several methods in a class require the same task, like a search or a swap, you should use helper methods. The reduce method in the Rational class on p. 126 is an example of such a method. Also, wherever possible you should enhance the readability of your code by using helper methods to break long methods into smaller tasks. The organization of code into different-methods is known as procedural abstraction, which encapsulates each task in a class in a separate method of the class. Procedural abstraction is an example of top-down development within a class.

- The process of breaking a long method into a sequence of smaller tasks is sometimes called **stepwise refinement**.

Implementing Methods

DATA ENCAPSULATION
- Instance variables and helper methods are generally declared as private, which prevents client classes from accessing them. Data encapsulation is when the data and methods of an object are combined in a class so that the data can be hidden.

Implementing Methods

STUB METHOD

- Sometimes it makes more sense in the development of a class to test a calling method before testing a method it invokes. A stub is a dummy method that stands in for a method until the actual method has been written and tested. A stub typically has an output statement to show that it was called in the correct place, or it may return some reasonable values if necessary.

Implementing Methods

ALGORITHM

- An algorithm is a precise step-by-step procedure that solves a problem or achieves a goal.

- Don't write any code for an algorithm in a method until the steps are completely clear to you.

Example 1

- A program must test the validity of a four-digit code number that a person will enter to be able to use a photocopy machine. The number is valid if the fourth digit equals the remainder when the sum of the first three digits is divided by seven.

- Classes in the program may include an IDNumber, the four-digit code; Display, which would handle input and output; and IDMain, the driver for the program. The data structure used to implement an IDNumber could be an instance variable of type int, or an instance variable of type String, or four instance variables of type int-one per digit, and so on.

Example 1

- A top-down design for the program that tests the validity of the number is reflected in the steps of the main method of IDMain:

```
Create Display
Read in IDNumber
Check validity
Print message
```

Example 1

- Each method in this design is tested before the next method is added to main. If the display will be handled in a GUI (graphical user interface), stepwise refinement of the design might look like this:

Create Display
  Construct a Display
  Create window panels
  Set up text fields
  Add panels and fields to window
Read in IDNumber
  Prompt and read
Check validity of IDNumber
  Check input
    Check characters
    Check range
  Separate into digits
  Check validity property
Print message
  Write number
  State if valid

NOTE

1. The IDNumber class, which contains the four-digit code, is responsible for the following operations:
   - Split value into separate digits
   - Check condition for validity

   The Display class, which contains objects to read and display, must also contain an IDNumber object. It is responsible for the following operations:
   - Set up display
   - Read in code number
   - Display validity message

   Creating these two classes with their data fields (instance variables) and operations (methods) is an example of data encapsulation.

2. The Display method readCodeNumber needs private helper methods to check the input: checkCharacters and checkRange. This is an example of procedural abstraction (the use of separate methods to implement each task) and data encapsulation (making the data private within the class).

NOTE

3. Initially the programmer had just an IDNumber class and a driver class. The Display class was added as a refinement, when it was realized that handling the input and message display was separate from checking the validity of the IDNumber. This is an example of top-down development (adding an auxiliary class to clarify the code).

4. The IDNumber class contains no data fields that are objects. It is therefore an independent class. The Display class, which contains an IDNumber data member, has a composition relationship with IDNumber (Display has-a IDNumber).

5. When testing the final program, the programmer should be sure to include each of the following as a user-entered code number: a valid four-digit number, an invalid four-digit number, an n-digit number, where n f. 4, and a "number" that contains a nondigit character. A robust program should be able to deal with all these cases.

Example 2

A program must create a teacher's grade book. The program should maintain a class list of students for any number of classes in the teacher's schedule. A menu should be provided that allows the teacher to:

- Create a new class of students.
- Enter a set of scores for any class.
- Correct any data that's been entered.
- Display the record of any student.
- Calculate the final average and grade for all students in a class.
- Print a class list, with or without grades.
- Add a student, delete a student, or transfer a student to another class.
- Save all the data in a file.

Identifying Classes

•Use the nouns in the specification as a starting point for identifying classes in the program.

•The nouns are: program, teacher, grade book, class list, class, student, schedule, menu, set of scores, data, record, average, grade, and file.

•Eliminate each of the following:

      program   (Always eliminate "program" when used in this context.)

      teacher   (Eliminate, because he or she is the user.)

      schedule   (This will be reflected in the name of the external file for each class, e.g., apcs_period3. dat.)

      data, record   (These are synonymous with student name, scores, grades, etc., and will be covered by these features.)

      class   (This is synonymous with class list.)

Identifying Classes

• The following seem to be excellent candidates for classes: GradeBook, ClassList, Student, and FileHandler. Other possibilities are Menu, ScoreList, and a GUI_Display.

• On further thought: Basic independent objects are Student, Menu, Score, and FileHandler.

• Group objects are ClassList (collection of students), ScoreList (collection of scores), and AllClasses (collection of ClassListS). The controlling class is the GradeBook. A Display class is essential for many of the grade book operations, like showing a class list or displaying information for a single student.

Relationships between Classes

- There are no inheritance relationships. There are many composition relationships between objects, however. The GradeBook has-a Menu, the ClassList has-a Student (several, in fact!), a Student has-a name, average, grade, list_of_scores, etc.

- The programmer must decide whether to code these attributes as classes or data fields.

Identifying Behaviors

- Use the verbs in the specification to identify required operations in the program. The verbs are: maintain <list>, provide <menu>, allow <user>, create <list>, enter <scores>, correct <data>, display <record>, calculate <average>, calculate <grade>, print <list>, add <student>, delete <student>, transfer <student>, and save <data>.

- You must make some design decisions about which class is responsible for which behavior.

- For example, will a ClassList display the record of a single Student, or will a Student display his or her own record? Who will enter scores-the GradeBook, a ClassList, or a Student? Is it desirable for a Student to enter scores of other Students? Probably not!

Decisions

- Here are some preliminary decisions. The GradeBook will provideMenu. The menu selection will send execution to the relevant object.

- The ClassList will maintain an updated list of each class. It will have these public methods: addStudent, deleteStudent, transferStudent, createNewClass, printClassList, printScores, and updateList. A good candidate for a helper method in this class is search for a given student.

- Each Student will have complete personal and grade information. Public methods will includesetName, getName, enterScore, correctData, findAverage, getAverage, getGrade, and displayRecord.

Decisions

- Saving and retrieving information is crucial to this program. The FileHandler will take care of openFileForReading, openFileForWriting, closeFiles, loadClass, and saveClass. The FileHandler class should be written and tested right at the beginning, using a small dummy class list.

- Score, ScoreList, and Student are easy classes to implement. When these are working, the programmer can go on to ClassList. Finally, the Display GUI class, which will have the GradeBook, can be developed. This is an example of bottom-up development.

Example 3

- A program simulates a game of Battleships, which is a game between two players, each of whom has a grid where ships are placed. Each player has five ships:

battleship	0 0 0 0 0
cruiser	0 0 0 0
submarine	0 0 0
destroyer	0 0
frigate	0

Example 3

- The grids of the players' fleets may look like this. Any two adjacent squares that are taken must belong to the same ship, i.e., different ships shouldn't "touch."

Player 1   Player 2

Example 3

- Each player's grid is hidden from the, other player. Players alternate "shooting" at each other's ships by calling out a position, a row and column number. A player must make an honest response, "hit" or "miss." If it's a hit, a player gets another turn.

- If the whole ship has been hit, the owner must say something like, "You sank my cruiser." Each player must keep track of hits and misses. The first player to sink his opponent's fleet is the winner.

Identifying Classes

- The nouns in the specification are program, game, players, grid, ship, battleship, cruiser, sub-marine, destroyer, frigate, square, position, opponent, row, column, turn, hits, misses, fleet, winner.

- Eliminate each of the following:

program	Always eliminate.
row, col	These are parts of a given position or square, more suitable as instance variables for a position or square object.
hits, misses	These are simply marked positions and probably don't need their own class.
turn	Taking a turn is an action and will be described by a method rather than a class.
opponent	This is another word for player.

- The following seem to be good candidates for classes: Player, Grid, Position, Ship, Battleship, Cruiser, Submarine, Destroyer, and Frigate. Additionally, it seems there should be a Game Manager and Display.

Relationship between Classes

- This program provides two examples of inheritance relationships. Each of the five ships **`is-a`** Ship, and shares common features, like `isHit`, `isSunk`, and array of positions.

- However, each has a unique name, length, and position in the grid. This means that Ship is a good candidate for an abstract class with abstract methods like `getLength`, `getName`, and `getPositions`, which depend on the kind of ship.

Relationship between Classes

- The second inheritance relationship is between the grids. There are two types of grids for each player: his own **FleetGrid** (the current state of his own ships) and his opponent's **HitGrid**, which keeps track of his hits and misses. Each of these grids is-a **Grid**. A grid is a candidate for an interface, with a list of methods like **getAdjacentNeighbors**, **getRightNeighbor**, etc. Each of **FleetGrid** and **HitGrid** would implement **Grid**.

- There are several composition relationships in this program. A Player has-a **HitGrid** and a **FleetGrid** and also has five ships. The **GameManager** has each of the two Player objects and also has-a Display. The Display has each of the grids.

Identifying Behaviors

- Use the verbs to identify key methods in the program: simulate <game>, place <ships>, shoot <at position>, call out <position>, respond <hit or miss>, sink <ship>, inform that <ship was sunk>, keep track of <hits or misses>, sink <opponent's fleet>, win <game>.

Identifying Behaviors

- You need to decide who will do what. There's no definitive way of implementing the program, but it seems clear that the **`GameManager`** should run the game and declare the winner. Should the **`GameManager`** also be in charge of announcing if a ship is sunk? It makes sense because the game manager can see both players' grids. Each player should keep track of his calls, so that he can make an intelligent next call and also respond "hit" or "miss." Will each player have a display? Or will the Display have both players? You have to set it up so that a player can't see his opponent's **`FleetGrid`**, but he can see his own and also a grid showing the state of the calls he has made.

- Should each player have a list of his ships, so he can keep track of the state of his fleet? And what about each ship in the fleet? Should a ship have a list of its positions, and should it keep track of if it's hit or sunk?
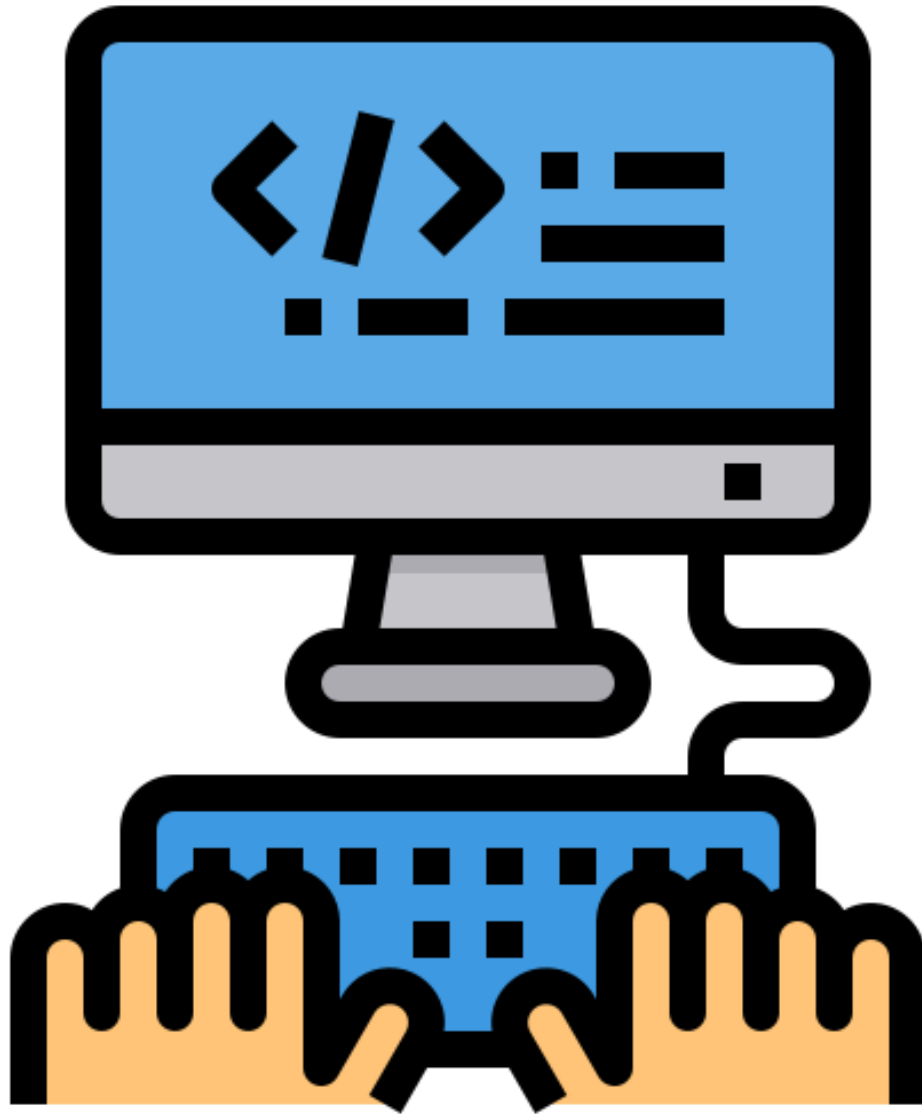
Identifying Behaviors

- Saving and retrieving updated information is crucial to this program. It seems a bit overwhelming. Where should you start? The Ship classes are low-level classes, independent of the players and grids. Start with these and test that you can get accurate information about each ship. In your driver program create an `ArrayList<Ship>`. Have a loop that prints information about each ship. Polymorphism will take care of getting the correct information about each ship.

- Now try the Grid classes. This is a complicated program where each small piece should be coded and tested with simple output. For example, a Grid can be displayed with a two dimensional array of 0's and 1's to show the positions of ships. Other symbols can be used to show what's been hit and what's been sunk.

Identifying Behaviors

- When everything is working with the grids, you could add a Display class that has Grid variables and a display method.

- Try a Player. Give him a list of ships, two grids and a Display.

- Then create a GameManager. Give her two Player variables and be sure she has a playGame method.

- The program development shown above is an example of bottom-up development.

| Vocabulary	Meaning
software development	Writing a program
object-oriented program	Uses interacting objects
program specification	Description of a task
program design	A written plan, an overview of the solution
program implementation	The code
test data	Input to test the program
program maintenance	Keeping the program working and up to date
top-down development	Implement main classes first, subsidiary classes later
independent class	Doesn't use other classes of the program in its code
bottom-up development	Implement lowest level, independent classes first
driver class	Used to test other classes; contains `main` method
inheritance relationship	*is-a* relationship between classes
composition relationship	*has-a* relationship between classes
inheritance hierarchy	Inheritance relationship shown in a tree-like diagram
UML diagram	Tree-like representation of relationship between classes
data structure	Java construct for storing a data field (e.g., array)
data encapsulation	Hiding data fields and methods in a class
stepwise refinement	Breaking methods into smaller methods
procedural abstraction	Using separate methods to encapsulate each task
algorithm	Step-by-step process that solves a problem
stub method	Dummy method called by another method being tested
debugging	Fixing errors
robust program	Screens out bad input
compile-time error	Usually a syntax error; prevents program from compiling
syntax error	Bad language usage (e.g., missing brace)
run-time error	Occurs during execution (e.g., `int` division by 0)
exception	Run-time error thrown by Java method
logic error	Program runs but does the wrong thing

Software Testing and Integration

Section 5

Testing and Debugging

TEST DATA

Not every possible input value can be tested, so a programmer should be diligent in selecting a representative set of test data. Typical values in each part of a domain of the program should be selected, as well as endpoint values and out-of-range values. If only positive input is required, your test data should include a negative value just to check that your program handles it appropriately.

UI Tests

End to End Tests

Integration Tests

Unit Tests

Service

Database

JS

Code
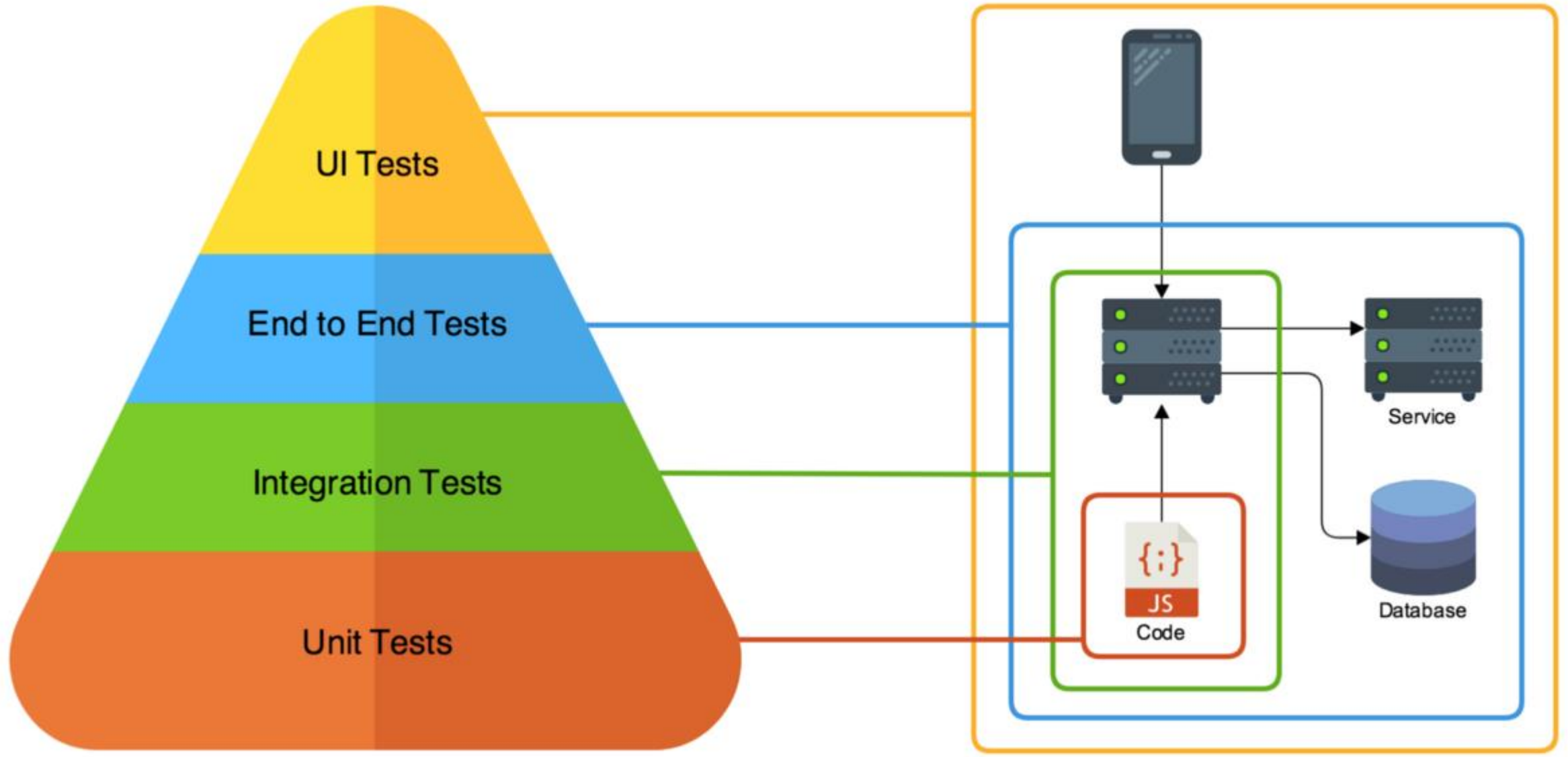
Example

- A program must be written to insert a value into its correct position in this sorted list:

    2   5   9

- Test data should include:
    - A value less than 2
    - A value between 2 and 5
    - A value between 5 and 9
    - A value greater than 9
    - 2, 5, and 9
    - A negative value

Types of errors

- There are 3 types of errors:
  - **Compiler error**: An error that the compiler will point out to you before you can run it
  - **Run-time error**: Will not be noticed by the compiler; will crash the program when it runs
  - **Logic Error**: does not cause either of the above errors, but makes your program run in a way you do not want it to

ALPHA TESTING?

BETA TESTING?

1. Documented business or software requirements

2. Test cases for all requirements

3. Testing team must have substantial knowledge about the software application

4. Test lab environment ready

5. QA build ready for execution

6. Test management tool ready to upload test cases and log defects

7. Traceability matrix ready to verify that each design requirement has at least one test case

1. Sign off on alpha testing

2. Beta version ready

3. Environment ready to release the product

4. Tool to capture real-time errors

5. Method of collecting user feedback

Robustness

- Always assume that any user of your program is not as smart as you are. You must therefore aim to write a robust program, namely one that:
  - Won't give inaccurate answers for some input data.
  - Won't crash if the input data are invalid.
  - Won't allow execution to proceed if invalid data are entered .
  - Examples of bad input data include out-of-range numbers, characters instead of numerical data, and a response of "maybe" when "yes" or "no" was asked for.
- Note that bad input data that invalidates a computation won't be detected by Java. Your program should include code that catches the error, allows the error to be fixed, and allows program execution to resume.

Program Maintenance and Version Control

Section 5

Program Maintenance

- Program maintenance involves upgrading the code as circumstances change.

- New features may be added. New programmers may come on board. To make their task easier, the original program must have clear and precise documentation.

Pre
Deployment

Post
Deployment

☑ **Check free space** 🗄

☑ **Inspect previous version** ✓

☑ **Find dependent files** 📁

🔗 **Create shortcut** ☑

⚙ **Manage services** ☑

📜 **Execute custom script** ☑

Centralized version control

Server

REPOSITORY

commit — update
commit — update
update — commit

WORKING COPY

WORKING COPY

WORKIN COPY

Workstation/PC #1   Workstation/PC #2   Workstation/PC #3
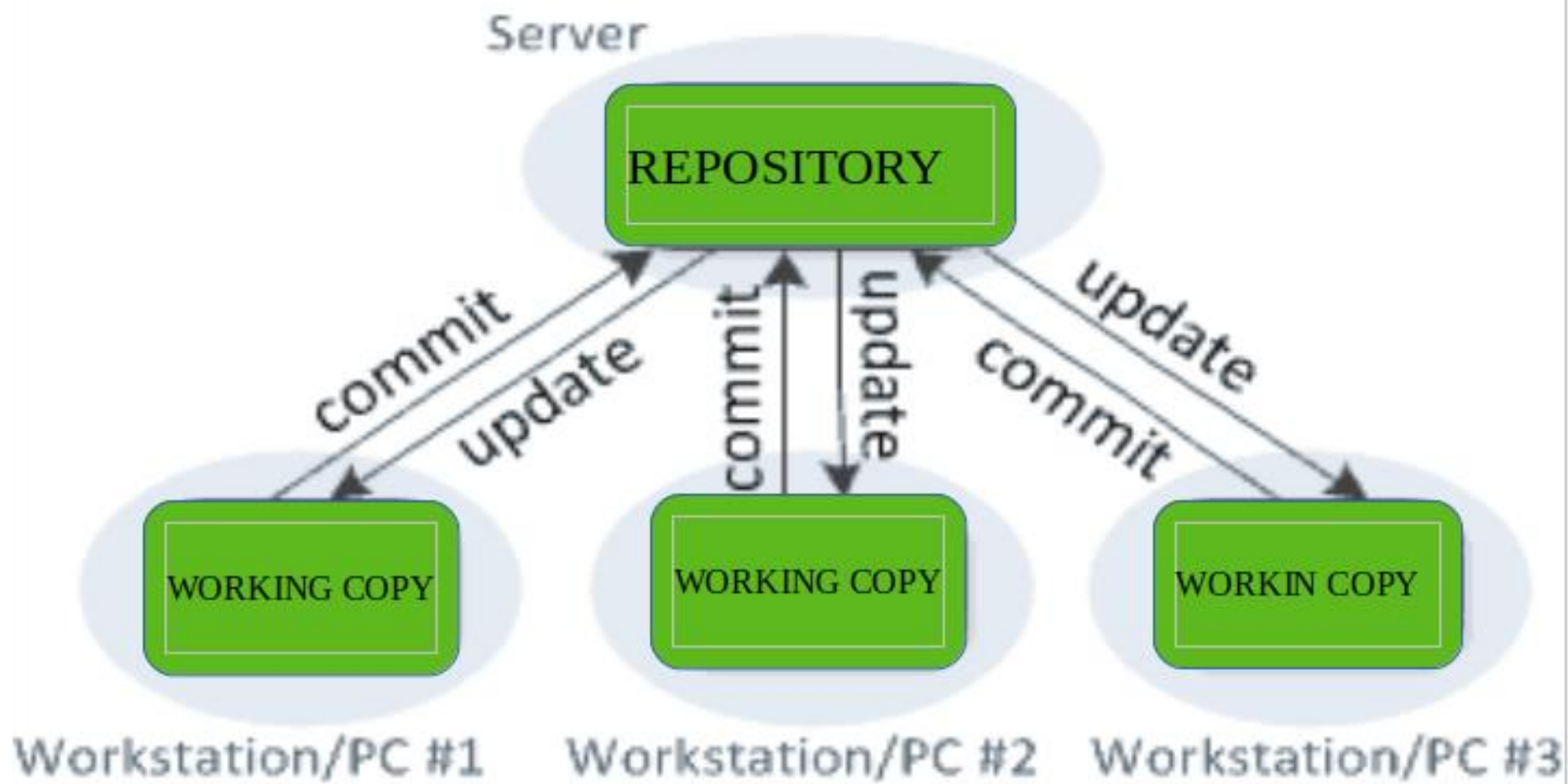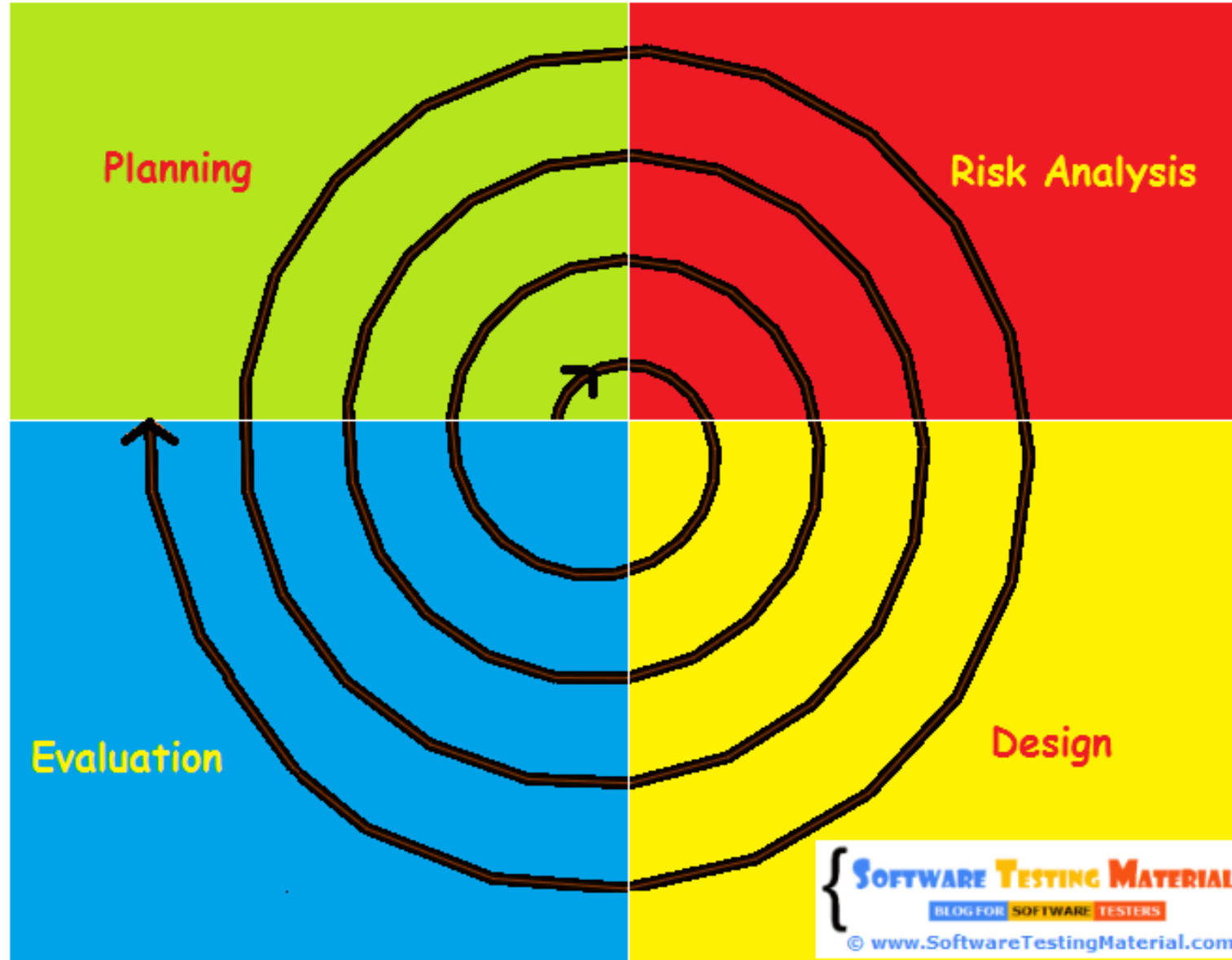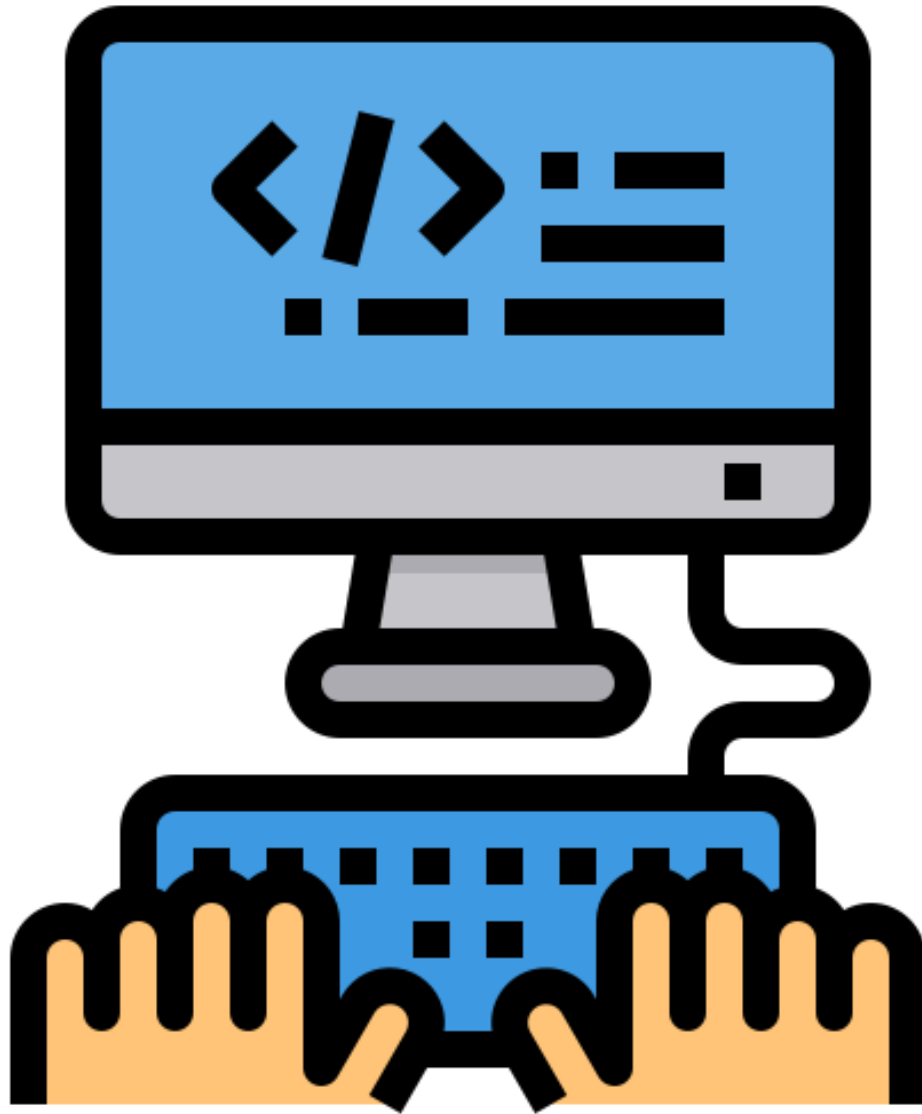
Software Development Tool and Flow

Section 5

Source code (developed by the programmer)

```
public class Welcome {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

Bytecode (generated by the compiler for JVM
to read and interpret)

```
...
Method Welcome()
  0 aload_0
  ...

Method void main(java.lang.String[])
  0 getstatic #2 ...
  3 ldc #3 <String "Welcome to Java!">
  5 invokevirtual #4 ...
  8 return
```

"Welcome to Java" is displayed on the console

```
Welcome to Java!
```

Create/Modify Source Code

Saved on the disk

Source Code

Compile Source Code
e.g., javac Welcome.java

If compile errors occur

Stored on the disk

Bytecode

Run Bytecode
e.g., java Welcome

Result

If runtime errors or incorrect result

Java Virtual Machine (JVM)
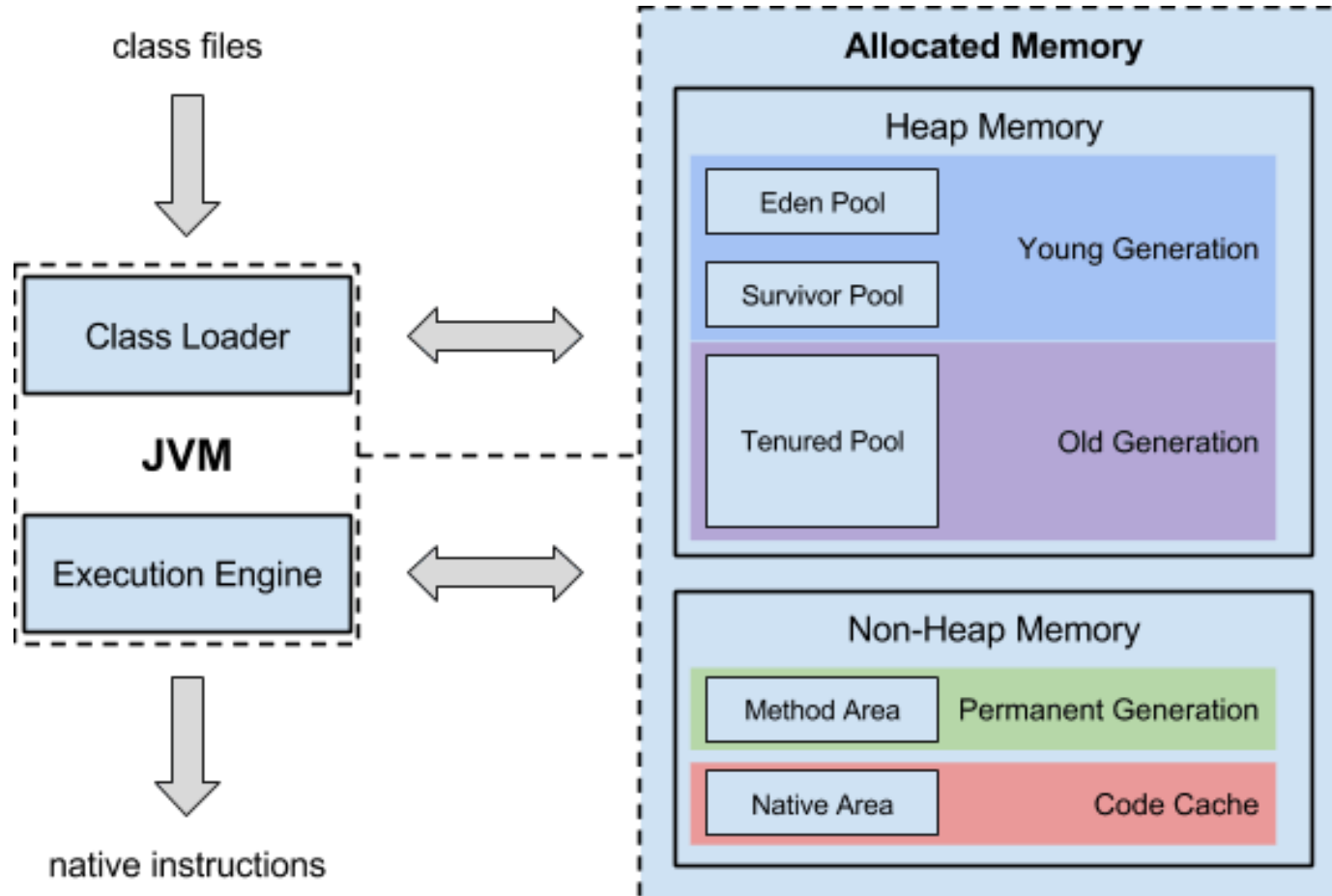
Program Analysis

Section 5

preconditions

input values

software
component

errors/exceptions

side effects

postconditions

output values
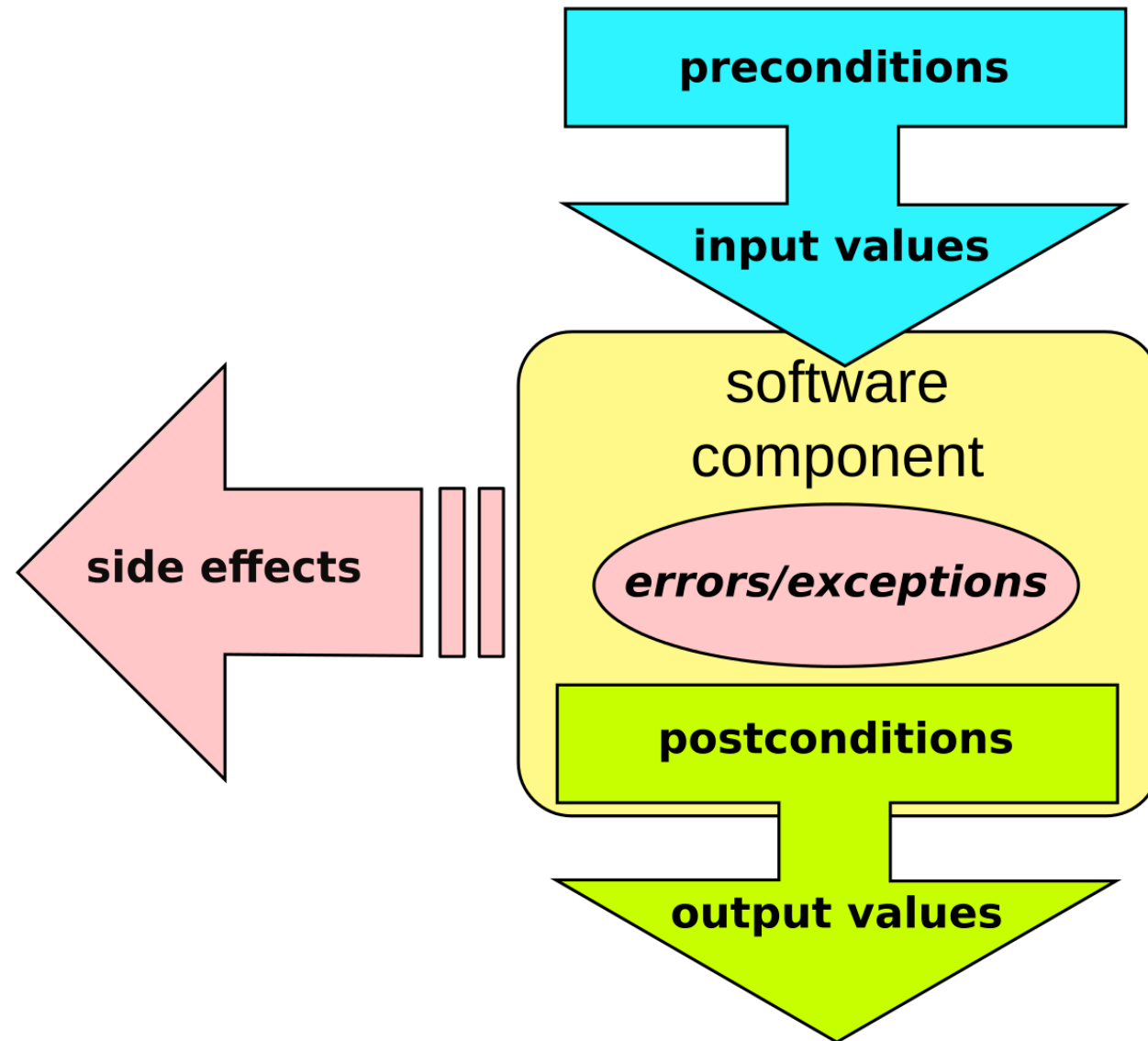
Program Correctness

- Testing that a program works does not prove that the program is correct. After all, you can hardly expect to test programs for every conceivable set of input data. Computer scientists have developed mathematical techniques to prove correctness in certain cases, but these are beyond the scope of the AP course.

- Nevertheless, you are expected to be able to make assertions about the state of a program at various points during its execution.

Assertions

- An assertion is a precise statement about a program at any given point. The idea is that if an assertion is proved to be true, then the program is working correctly at that point.

- An informal step on the way to writing correct algorithms is to be able to make different kinds of assertions about your code.

Assertions

- **PRECONDITION**
  - The precondition for any piece of code, whether it is a method, loop, or block, is a statement of what is true immediately before execution of that code.

- **POSTCONDITION**
  - The postcondition for a piece of code is a statement of what is true immediately after execution of that code.

Efficiency

- An efficient algorithm is one that is economical in the use of:
  - CPU time. This refers to the number of machine operations required to carry out the algorithm (arithmetic operations, comparisons, data movements, etc.).
  - Memory. This refers to the number and complexity of the variables used.
- Some factors that affect run-time efficiency include unnecessary tests, excessive movement of data elements, and redundant computations, especially in loops.

Efficiency

- Always aim for early detection of output conditions: ¥our sorting algorithm should halt when the list is sorted; your search should stop if the key element has been found.

- In discussing efficiency of an algorithm, we refer to the best case, worst case, and average case. The best case is a configuration of the data that causes the algorithm to run in the least possible amount of time. The worst case is a configuration that leads to the greatest possible run time. Typical configurations (i.e., not specially chosen data) give the average case. It is possible that best, worst, and average cases don't differ much in their run times.

Efficiency

- For example, suppose that a list of distinct random numbers must be searched for a given key value. The algorithm used is a sequential search starting at the beginning of the list. In the best case, the key will be found in the first position examined. In the worst case, it will be in the last position or not in the list at all.

- On average, the key will be somewhere in the middle of the list.

One Dimensional Array

Section 6
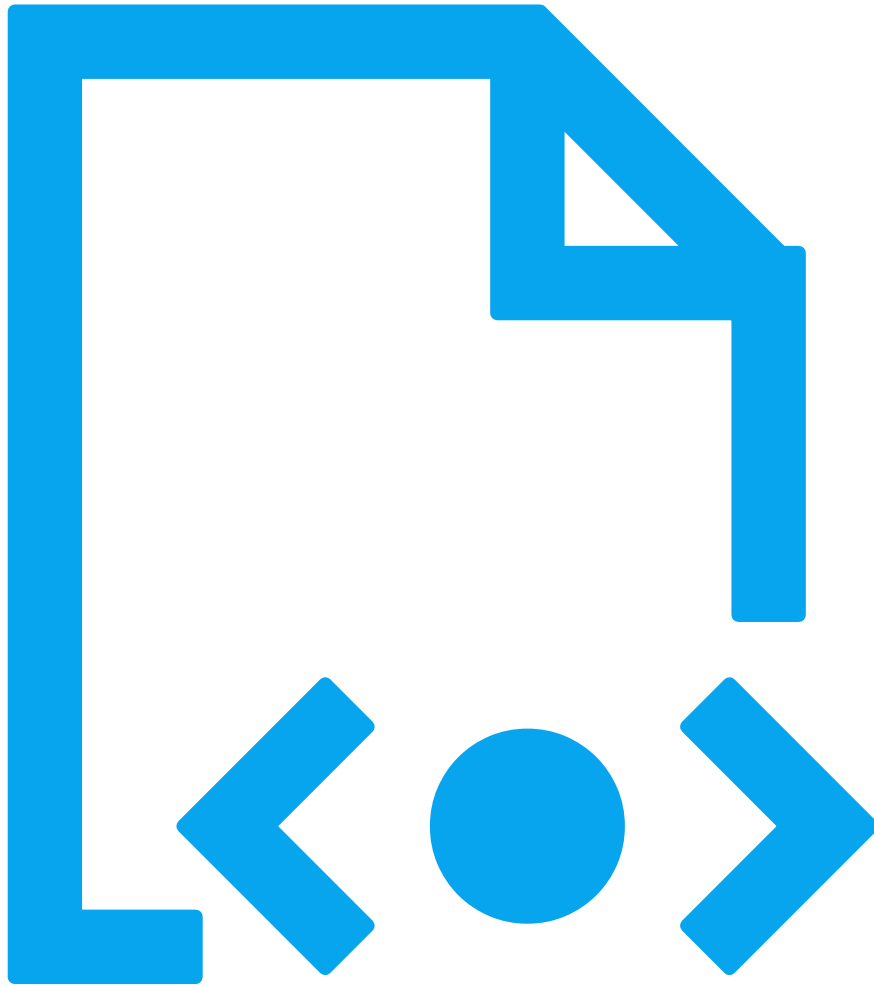
Reference Type

INDIRECT ACCESS
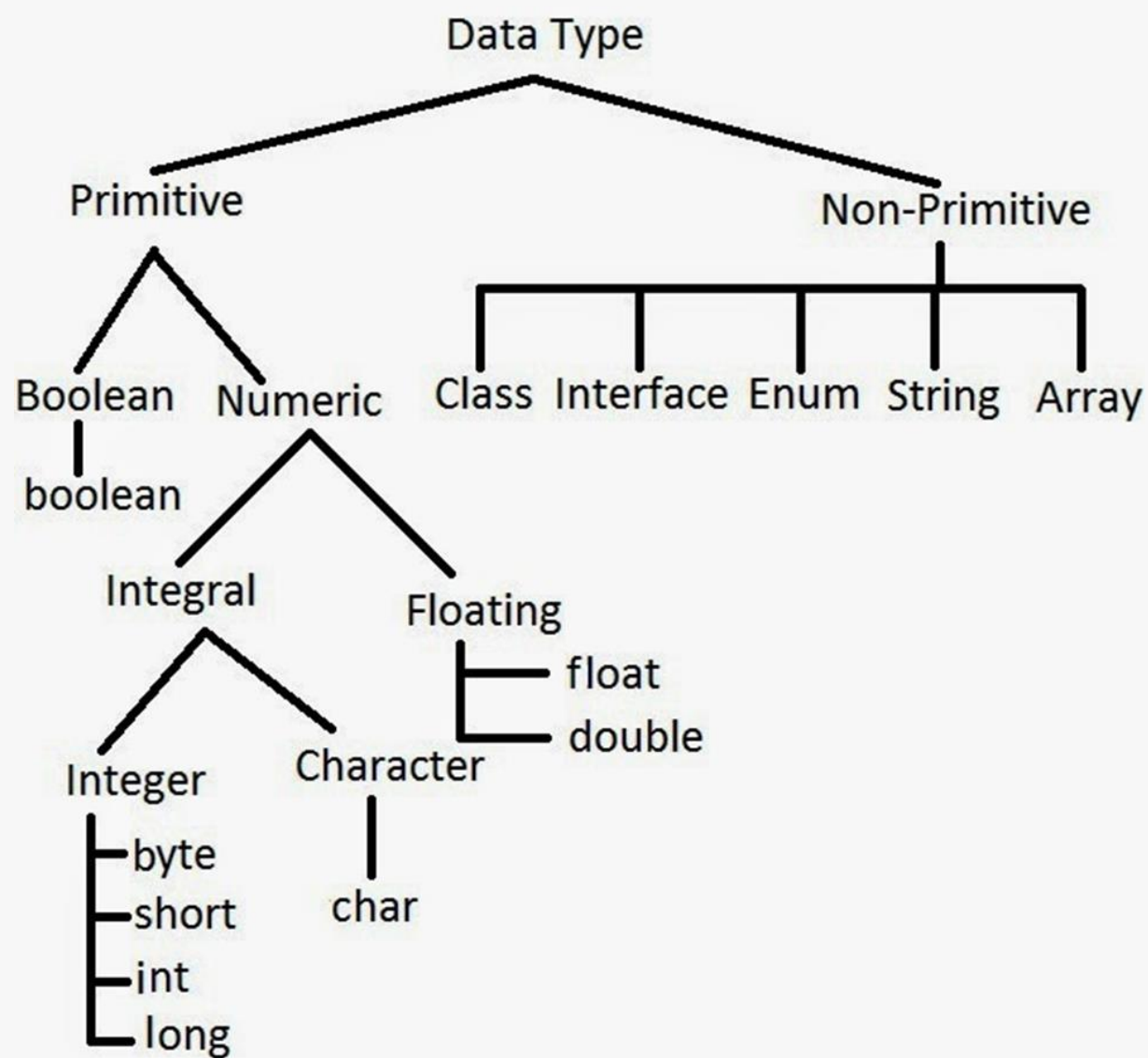
Reference Date Type
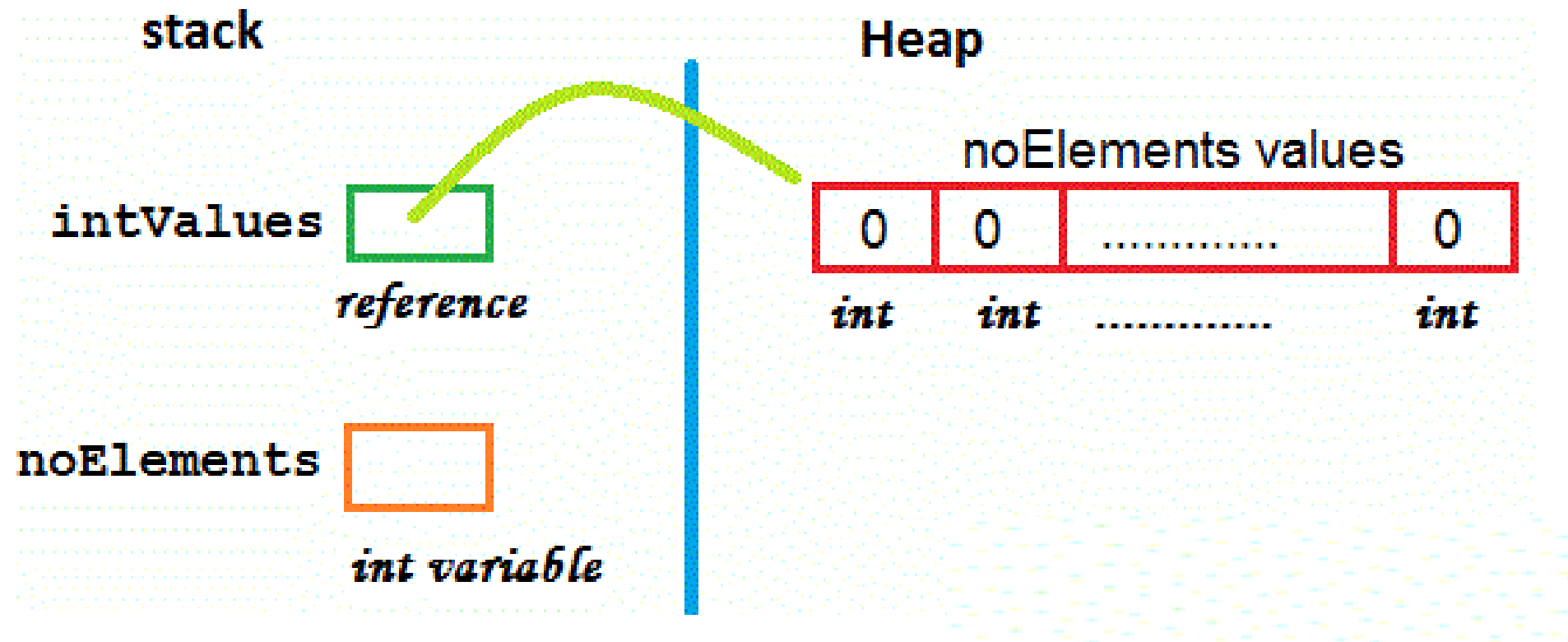
Three Important Complex Data Structures

There are five basic data structures (ADT) that are tested on the AP Computer Science A Exam:

- String (Covered in Week 2)
- The array (of one-dimensional array)
- The 2-D array
- The ArrayList
- Class (Structure, Data Record)

Situation	Recommendation	Justification
A program helps an elevator know what floor it is on.	Array	The number of floors is fixed. The elevator can go to any floor by knowing what number it is.
Facebook keeps track of how many friends you have.	ArrayList	The number of friends you have on Facebook may increase or decrease. You are allowed to add or remove anyone at any time regardless of where they are in the list.
Your program is going to simulate chess, Candy Crush, or 2048	2-D Array	Each of these games can be simulated on either a square or rectangular grid in which the row and column are used to find out what is in each cell.
A cell phone keeps track of text messages	ArrayList	The number of text message on a cell phone can increase or decrease. You can even delete all of the messages.
A program keeps track of classes you have each period of the school day.	Array	The number of class periods in the school day is fixed. Each period is assigned a value
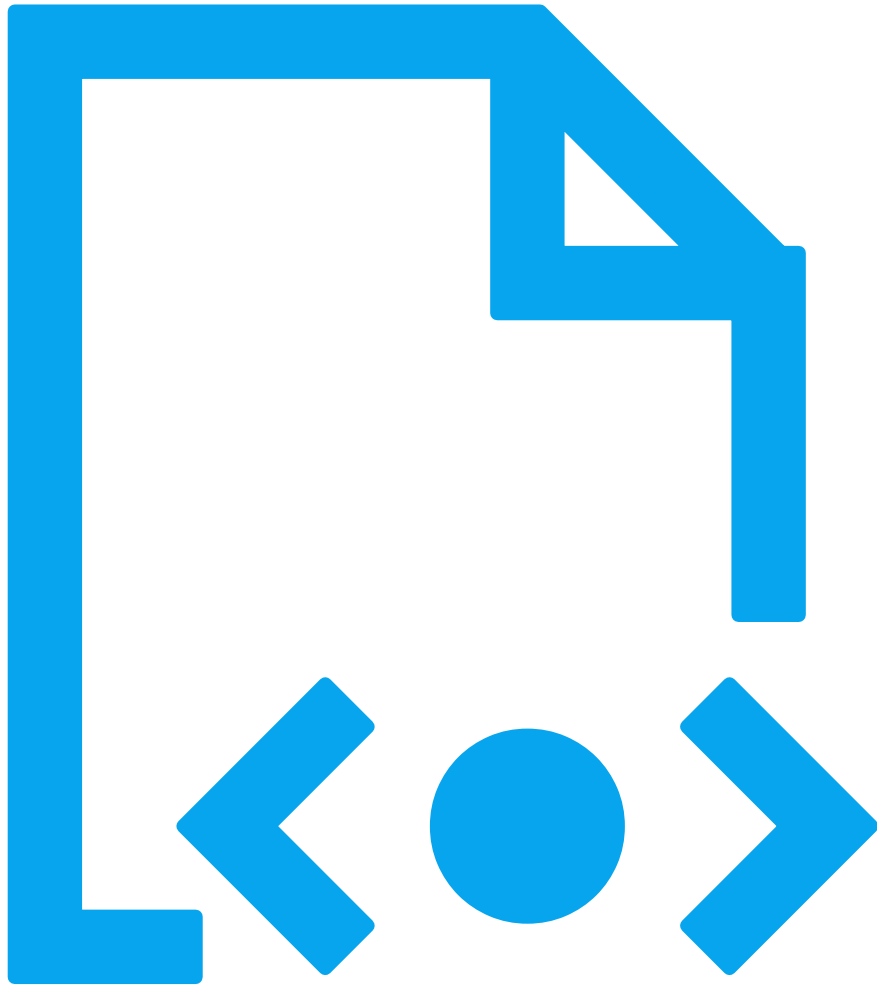
Array Definition

WHAT IS ARRAY?

Introducing Arrays

- Array is a data structure that represents a collection of the same types of data.

double[] myList = new double[10];

myList[0]	5.6
myList[1]	4.5
myList[2]	3.3
myList[3]	13.2
myList[4]	4
myList[5]	34.33
myList[6]	34
myList[7]	45.45
myList[8]	99.993
myList[9]	11123

myList → reference

Array reference variable

Array element at index 5

Element value
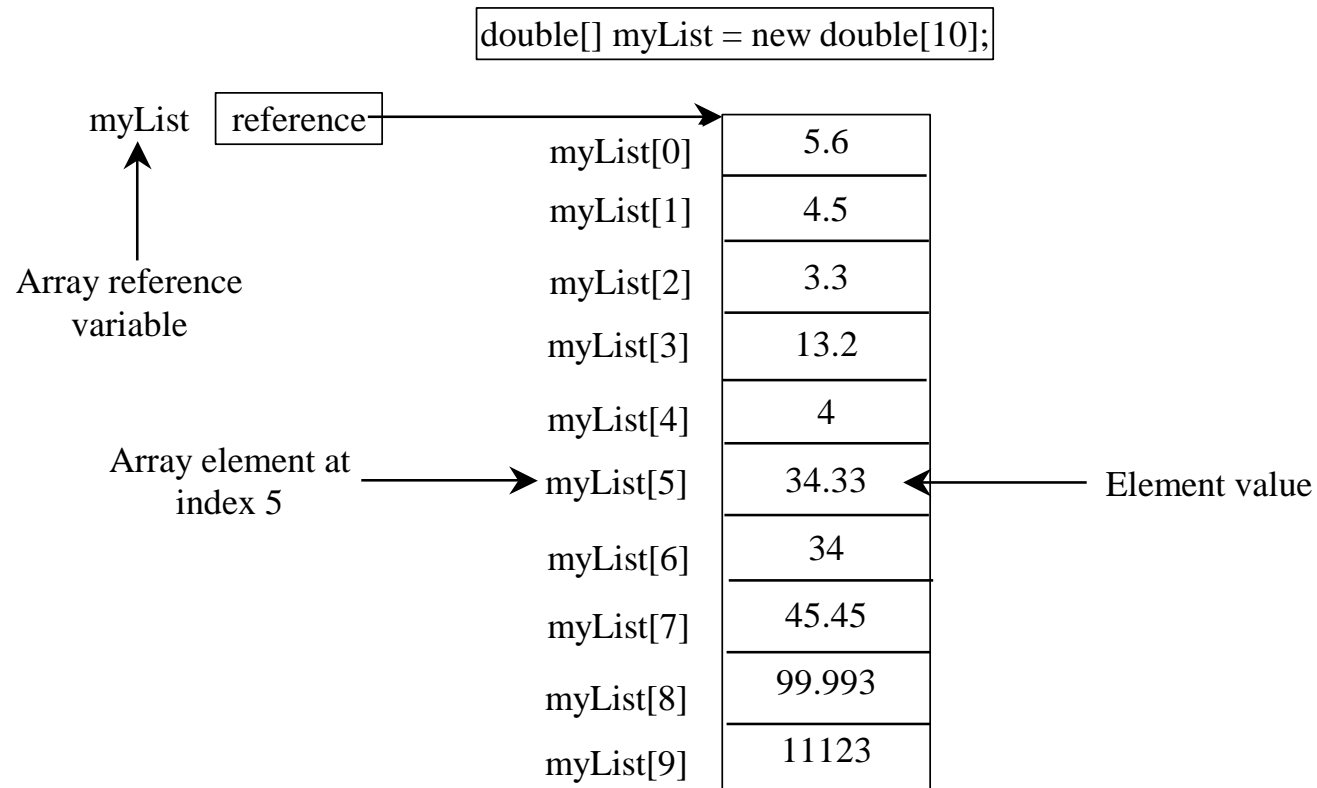
Pre-defined Array

- **Pre-defined Array:**
  int[] intArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

- **Array Declaration with array memory allocation:**
  int[] intArray2 = new int[10];

- **Array declaration and data assignment with anonymous array:**
  int[] intArray3;
       intArray3 = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

array.length
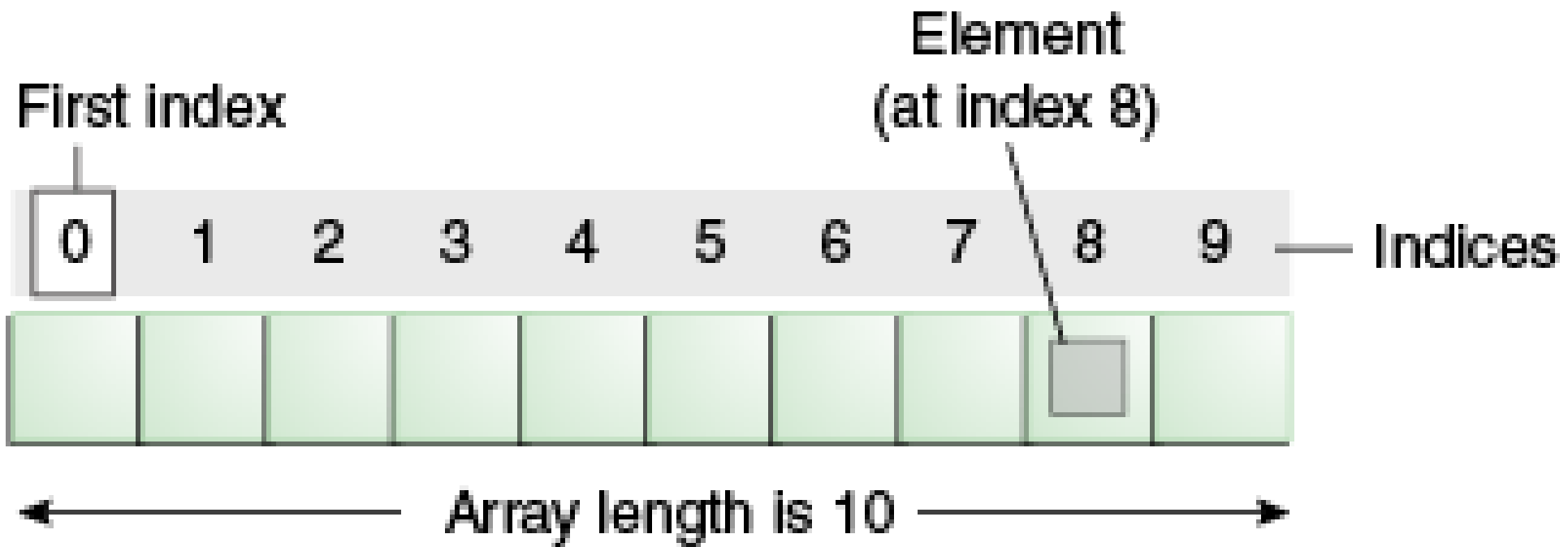
The **new** Operator

- An object of a class is named or declared by a variable of the class type:

  ClassName classVar;

- The new operator must when be used to create the object and associate it with its variable:

  classVar = **new** ClassName();

- These can be combined as follows:

  className classVar = **new** ClassName();

new Operator
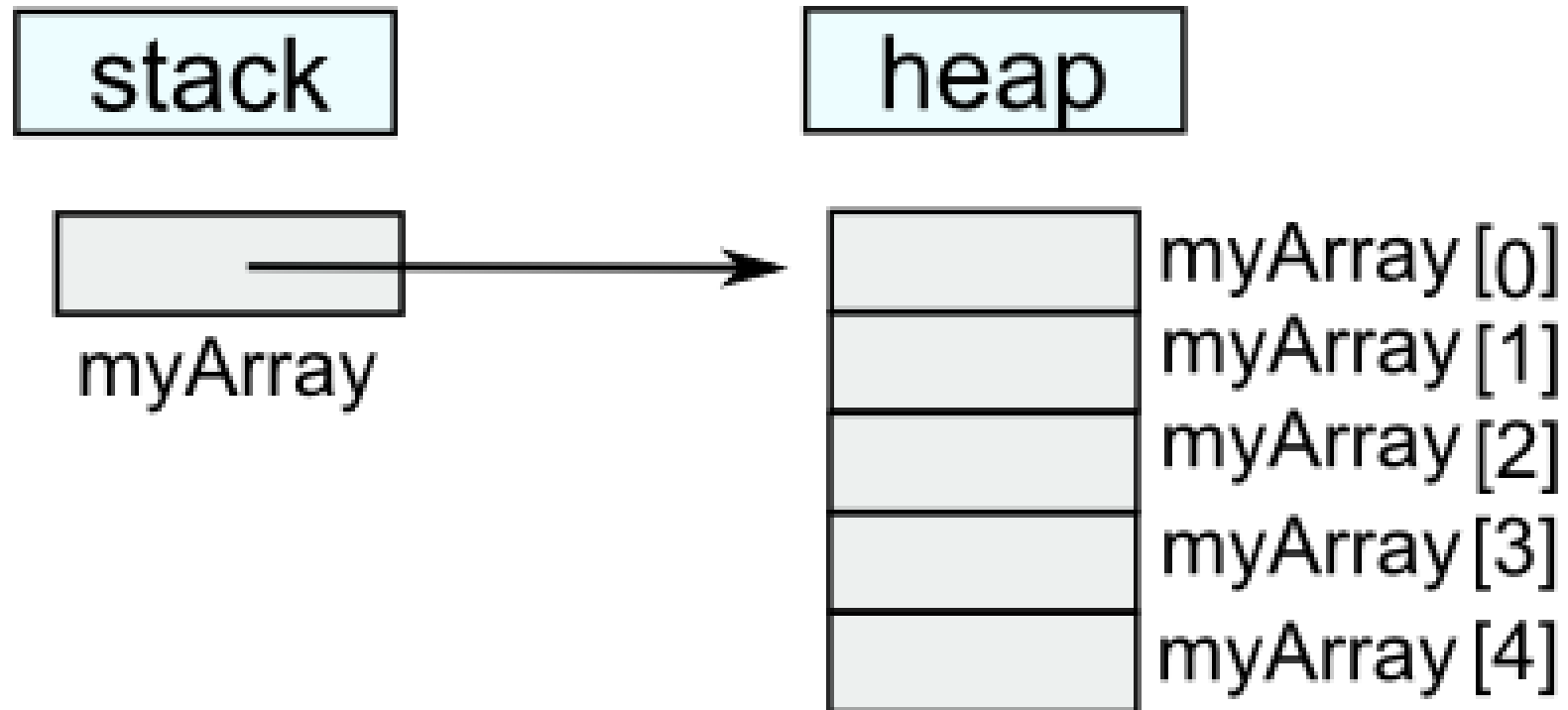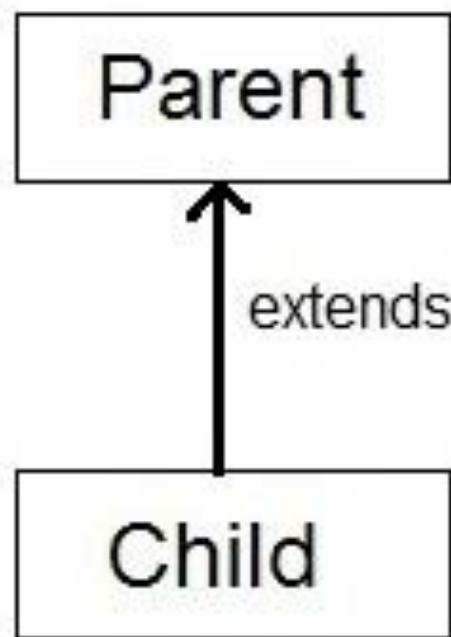
Java Heap

String s1 = "Cat";

String s2 = "Cat";

String s3 = new String("Cat");

s1 == s2; //true

s1 == s3; //false

"Cat"

"Dog"

"Cat"

String Pool
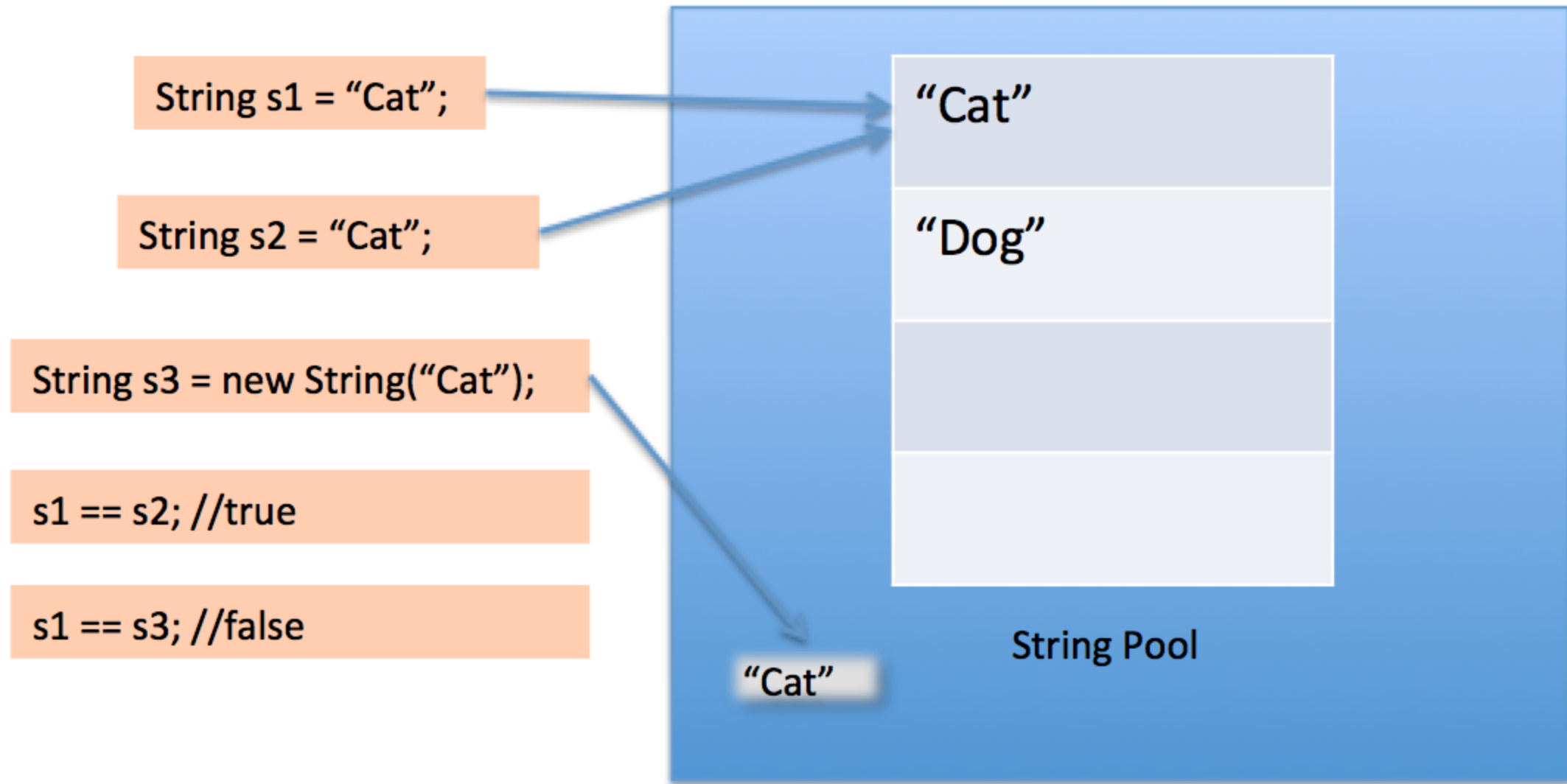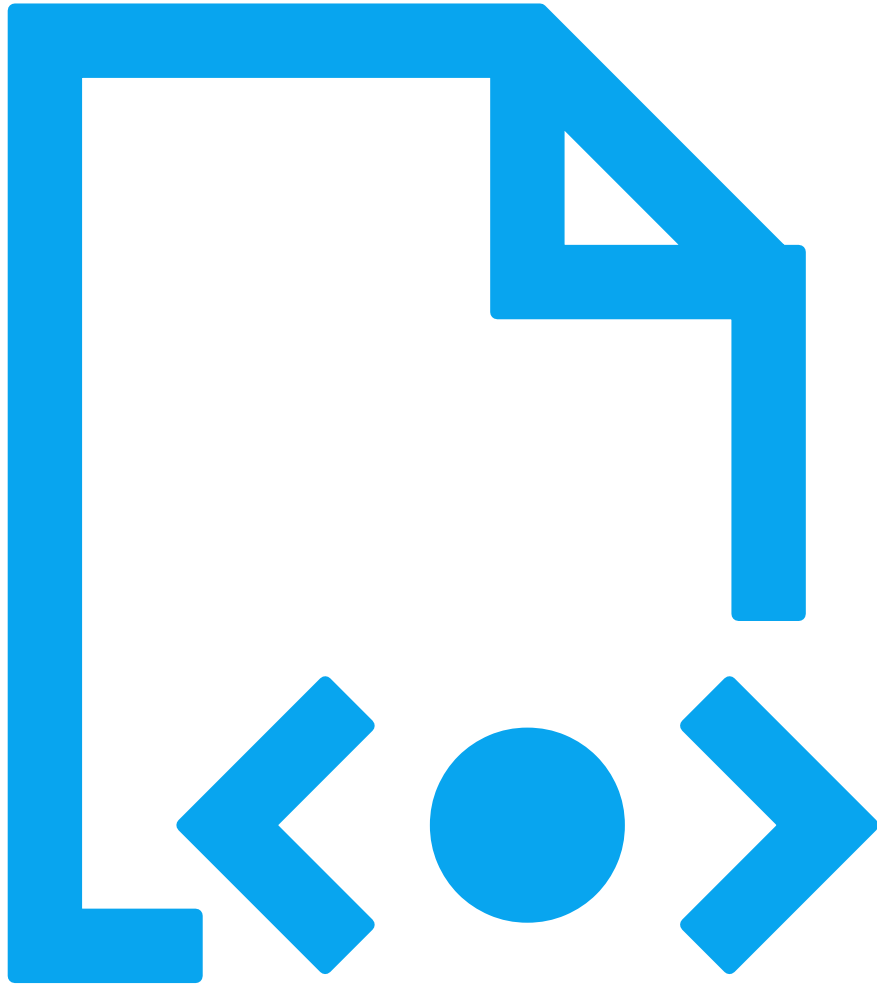
Basic Array Class

BASIC APPLICATION AND INDEXING

| *create an array with random values* | ```java
double[] a = new double[n];
for (int i = 0; i < n; i++)
    a[i] = Math.random();
``` |
| *print the array values, one per line* | ```java
for (int i = 0; i < n; i++)
    System.out.println(a[i]);
``` |
| *find the maximum of the array values* | ```java
double max = Double.NEGATIVE_INFINITY;
for (int i = 0; i < n; i++)
    if (a[i] > max) max = a[i];
``` |
| *compute the average of the array values* | ```java
double sum = 0.0;
for (int i = 0; i < n; i++)
    sum += a[i];
double average = sum / n;
``` |
| *reverse the values within an array* | ```java
for (int i = 0; i < n/2; i++)
{
    double temp = a[i];
    a[i] = a[n-1-i];
    a[n-i-1] = temp;
}
``` |
| *copy sequence of values to another array* | ```java
double[] b = new double[n];
for (int i = 0; i < n; i++)
    b[i] = a[i];
``` |

Array as Stack
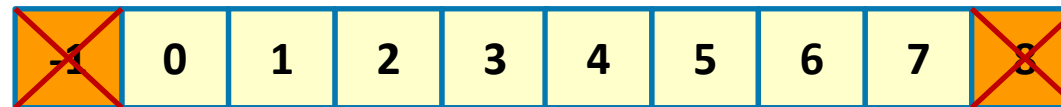## Demo Program: StackOfIntegers.java

Go BlueJ!!!

- **`size`** is also used as the top pointer for the stack.

Discrete Space
(last included, end not included)

Count = lastIndex-firstIndex+1 ;

Count = endIndex-startIndex;

Array with 8 elements (indexed from 0 to 7)

-1	0	1	2	3	4	5	6	7	8

8-0 (=7-0+1) = 7- (-1)

Number of elements is 8-0 or 7 – (-1).
Use 7 – (-1)   (Last element – the element before first element)
Or, 8-0   (The empty element – the first element)

Base and local index

	-8	-7	-6	-5	-4	-3	-2	-1	0
$n\text{-}i$									✕

	4	5	6	7	8	9	10	11	12
$b\text{+}i$									✕

	0	1	2	3	4	5	6	7	8
i									✕

Index = Base + local_Index
$$= b\text{+}i$$

Index = Length - local_neg_Index
$$= n\text{-}i$$

Base = b = 4, Length = n = $a.length$

Indexing for ASCII code

- Total number of ASCII letters: 'Z'-'A'+1 (alphabet size)

- Indexing for a letter ('B'): 'A' + ('B'-'A')

- Traversing through the whole alphabet: 'A' + i;   i is from 0 to 25

- Letter indexing is also used for histogram: a['X'-'A'] to store the number of occurrence for the letter 'X'

Sum up an array of 8 elements with index from the last element to the first.

- Write a program to sum up

int a = {1, 2, 3, 4, 5, 6, 7, 8};

- starting from 8 down to 1 with proper indexing.

```java
public static void main(String[] args){
    int[] a = {1, 2, 3, 4, 5, 6, 7, 8};

    int sum=0;
    for (int i=a.length-1; i>=0; i--){
        sum+= a[i];
    }
    System.out.println(Arrays.toString(a)+"'s sum="+sum);
}
```

CopyTo(src, dest, fromIndex) Method

Demo Program: CopyTo.java

```java
static int[] a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
static int[] b = {10, 11, 12};

public static void copyTo(int[] source, int[] destination, int fromIndex){
    if (fromIndex+source.length > destination.length) return;
    for (int i=0; i<source.length; i++){
        destination[fromIndex+i] = source[i];
    }
}
```

Key Methods in Arrays

Key Methods in java.util.Arrays	Descriptions
static List **asList**(T[])	Convert an array to a List (and bind them)
static int **binarySearch**(Object[], key) static int **binarySearch**(primitive[], key)	Search a sorted array for a given value, return an index or insertion point
static int **binarySearch**(T[], key, Comparator)	Search a Comparator-sorted array for a value
static boolean **equals**(Object[], Object[]) static boolean **equals**(primitive[], primitive[])	Compare two arrays to determine if their contents are equal
public static void **sort**(Object[ ]) public static void **sort**(primitive[ ])	Sort the elements of an array by natural order
public static void **sort**(T[], Comparator)	Sort the elements of an array using a Comparator
public static String **toString**(Object[]) public static String **toString**(primitive[])	Create a String containing the contents of an array

(Non-AP: Use with cautions)
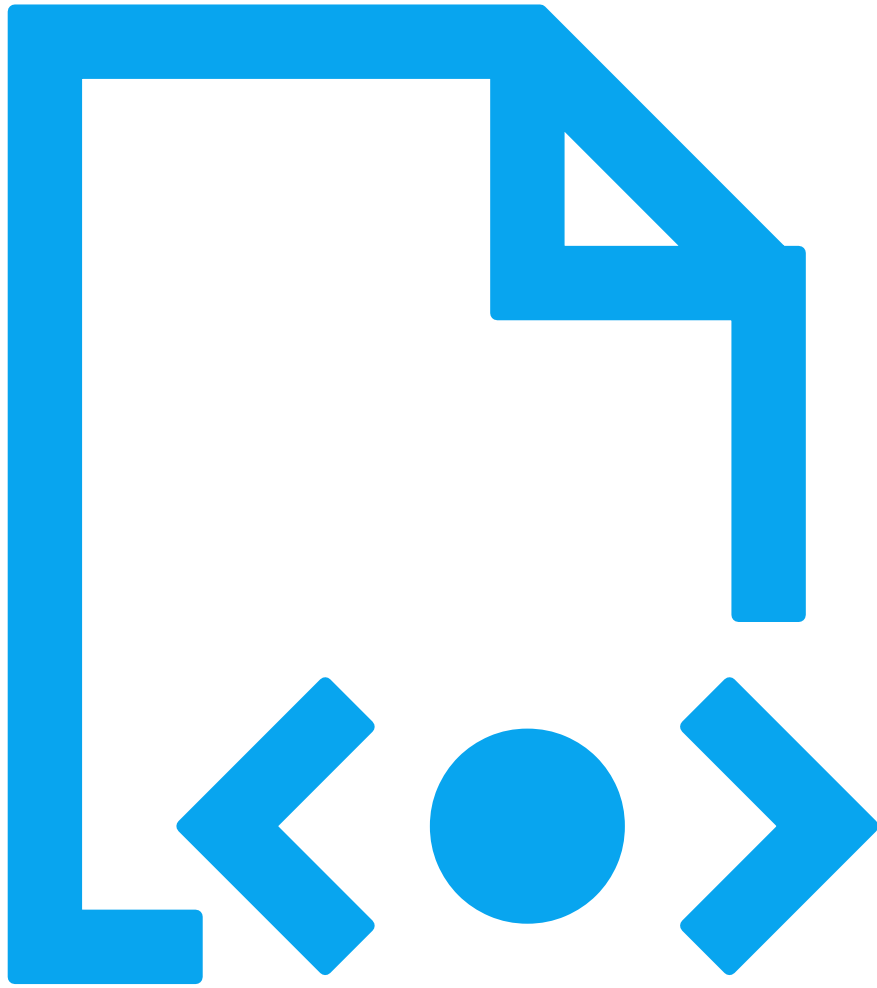
For Each Loop

ITERABLE ITEMS ONLY

Enhanced for-loop

General Form for the Enhanced for loop (the for-each loop):

datatype[] arrayName = /* array filled in some way */

for (datatype temporaryVarible: arrayName){

   // instructions that sue temporaryVariable

}

What data structure can be accessed by for-each loop?

Any container class that implements both Iterable Interface and Iterator Interface can be accessed by for-each loop.

Iterable Interface:

 iterator() method

Iterator Interface:

hasNext() method

next() method

remove() method

```java
import java.lang.Iterable;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Arrays;
public class StringList implements Iterable<String>, Iterator<String>{
    String[] list = new String[10];
    int length=0;
    int i = 0;
    StringList(String[] source){
        list = source;
        length = list.length;
    }
    public int size(){ return length; }
    public Iterator<String> iterator(){
        return this;
    }
    public String toString(){
        return Arrays.toString(list);
    }
    public boolean hasNext(){
        return i<length;
    }
    public String next(){
        if (!hasNext()) throw new NoSuchElementException();
        return list[i++];
    }
    public void remove(){
        throw new UnsupportedOperationException();
    }
    public void reset(){
        i=0;
    }
}
```

```java
public static void main(String[] args){
    String[] s = {"alpha", "beta", "gamma", "delta", "epsilon"};
    StringList alist = new StringList(s);
    //System.out.println(alist);
    //System.out.println(alist.size());
    for (String str: alist){
        System.out.println(str);
    }
    for (String str: alist){
        System.out.println(str);
    }
}
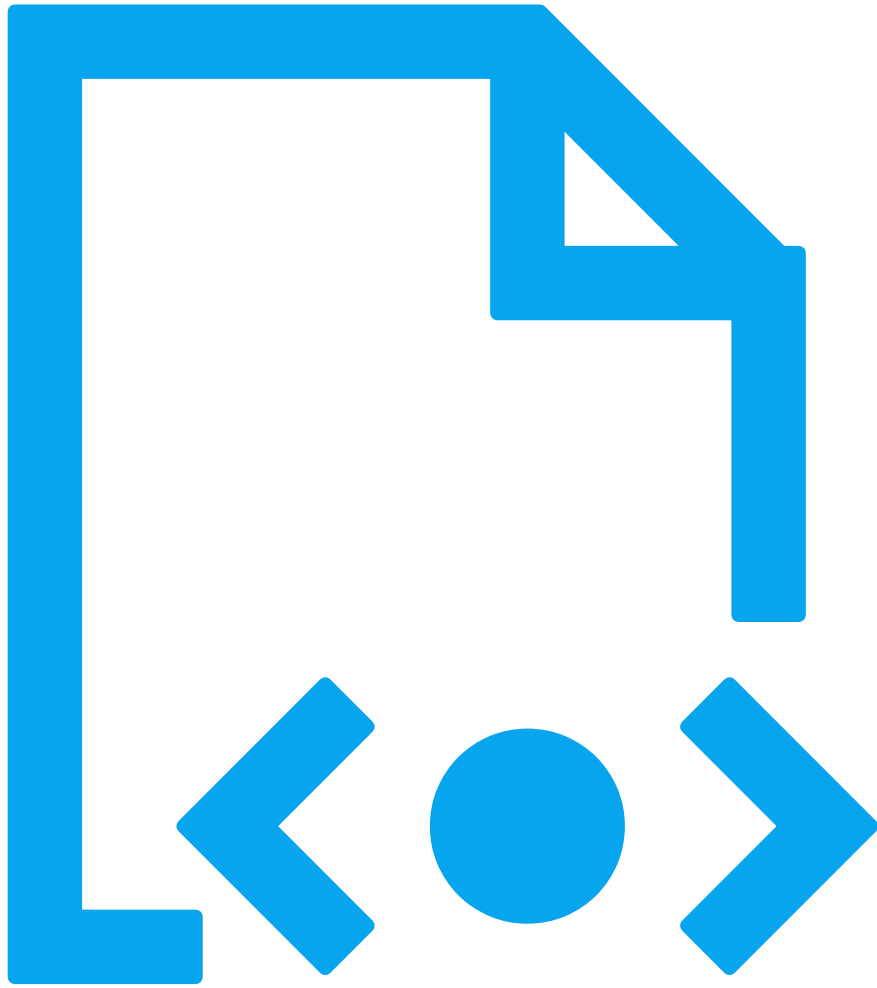```

BlueJ: Terminal Window - Week3

Options

```
alpha
beta
gamma
delta
epsilon
alpha
beta
gamma
delta
epsilon
```
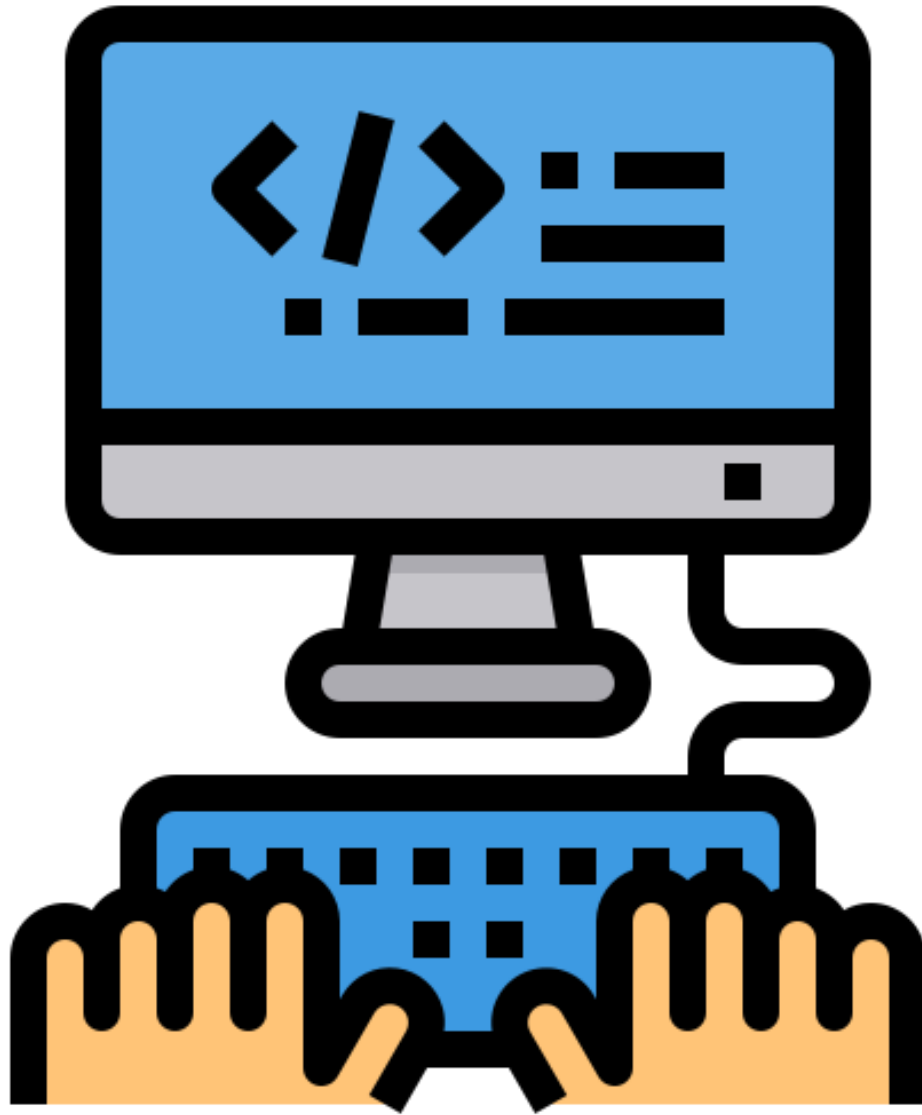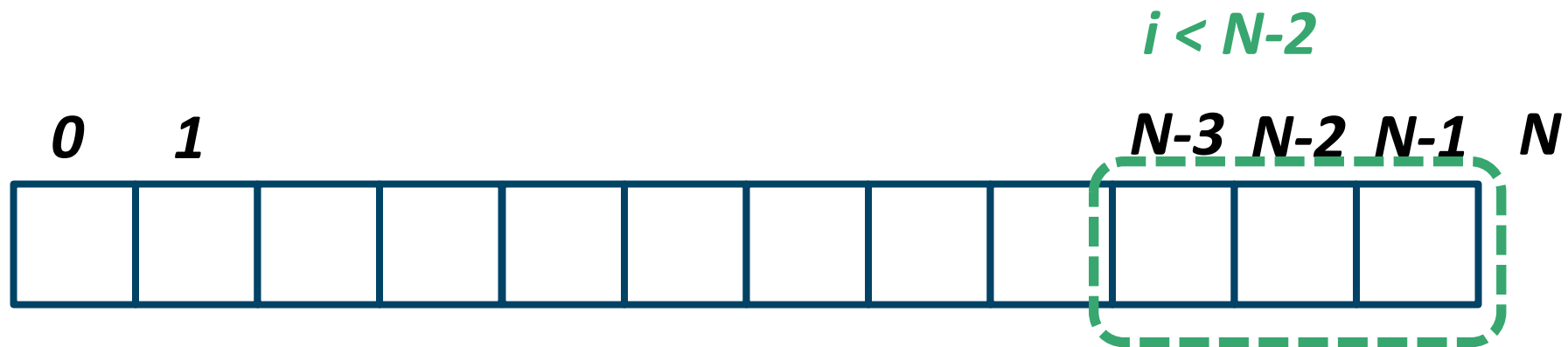
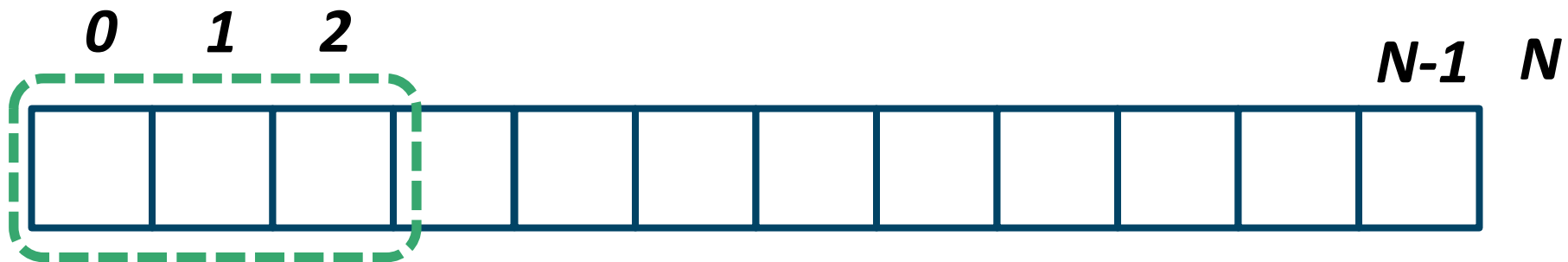# Array Processing

Section 6

# Array Indexing

HOW TO CALCULATE?

# Window of 3

```
for (int i=0; i<arr.length-2; i++){

    …

}
```
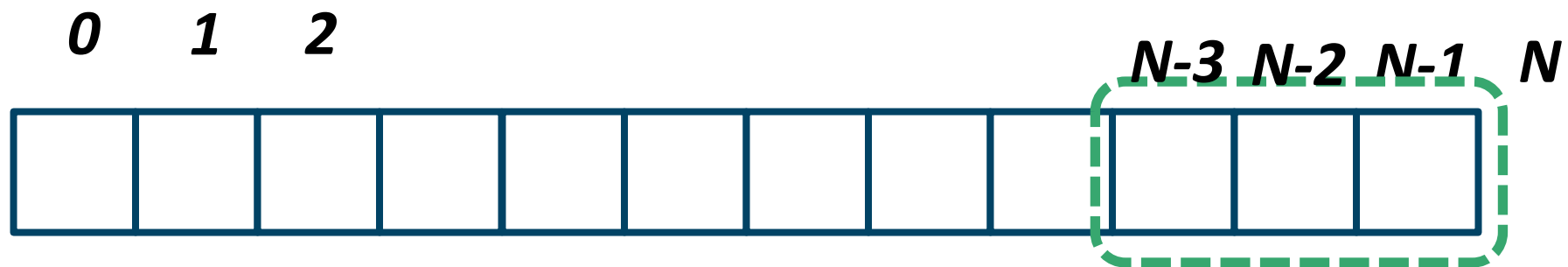
**i < N-2**

**0**    **1**                                          **N-3  N-2  N-1**    **N**

# First 3

```
for (int i=0; i<3; i++){

    …

}
```

Last 3

```
for (int i=N-3; i<N; i++){

    …

}
```

Middle by N/2
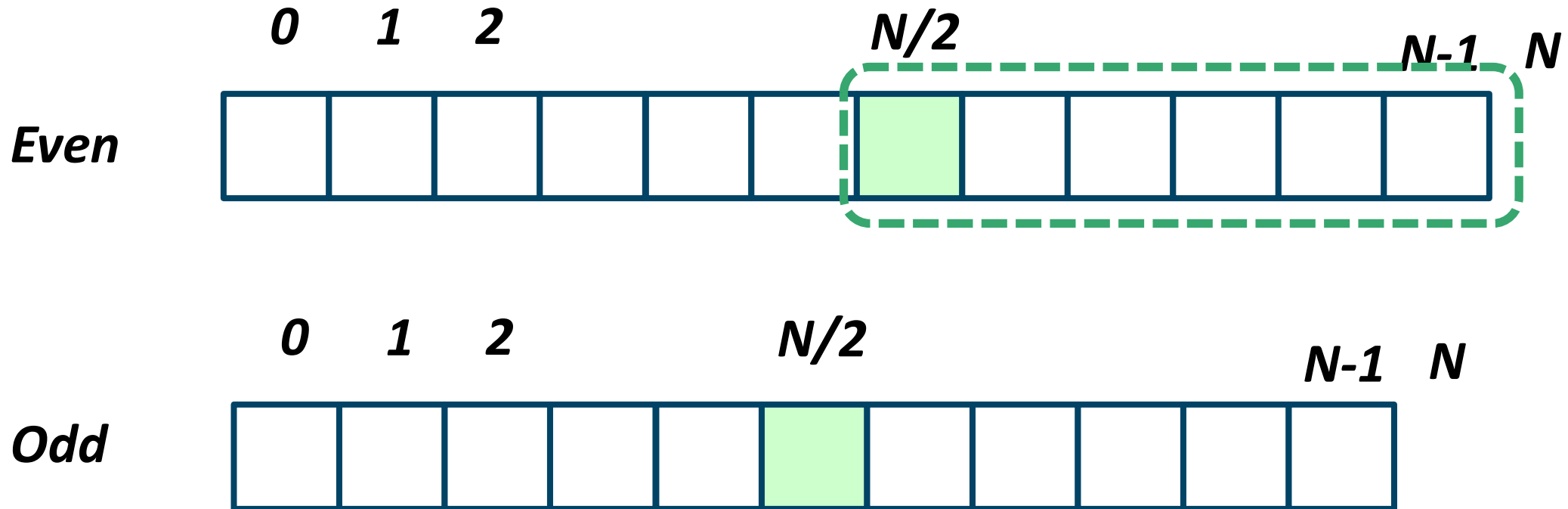
Middle by (low+high)/2

Middle by N/2

0   1   2   N/2   N-1   N

Even

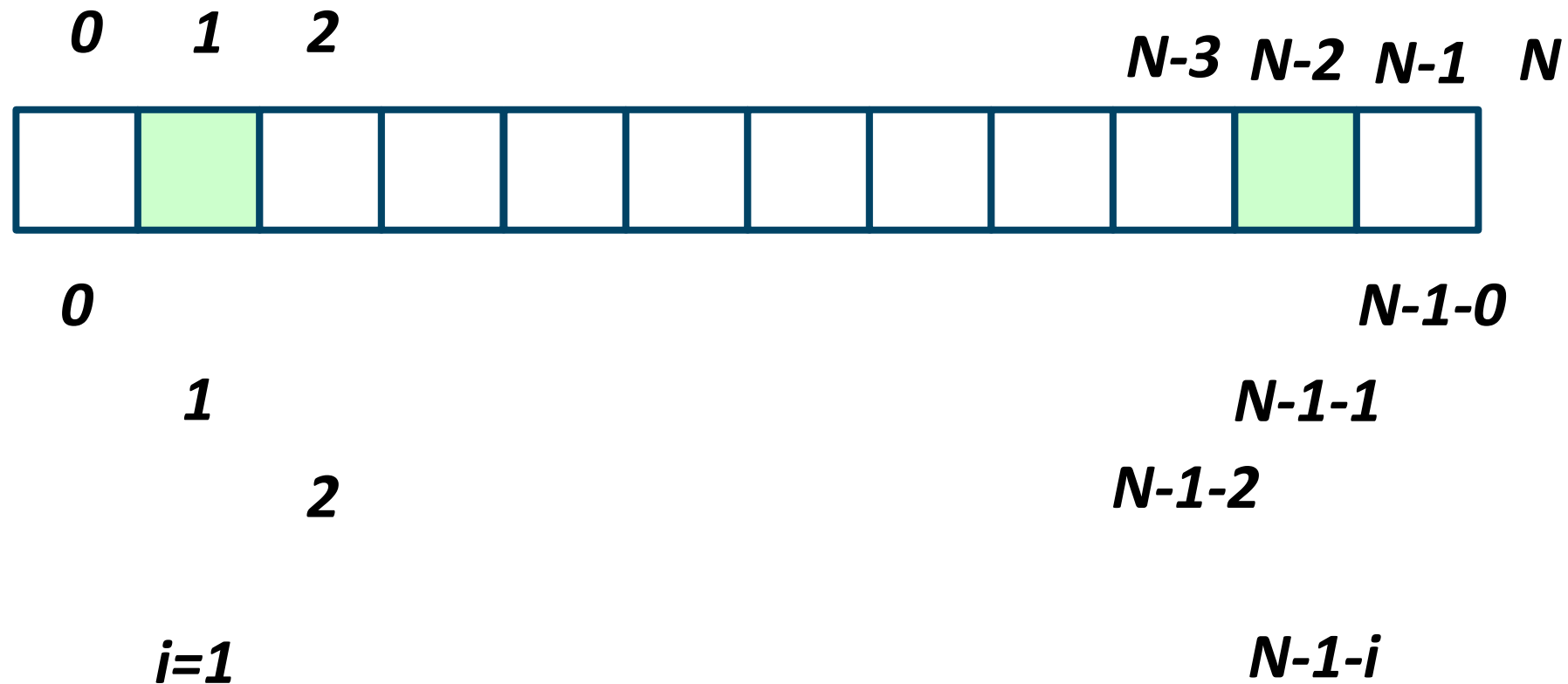**0   1   2   N/2   N-1   N**
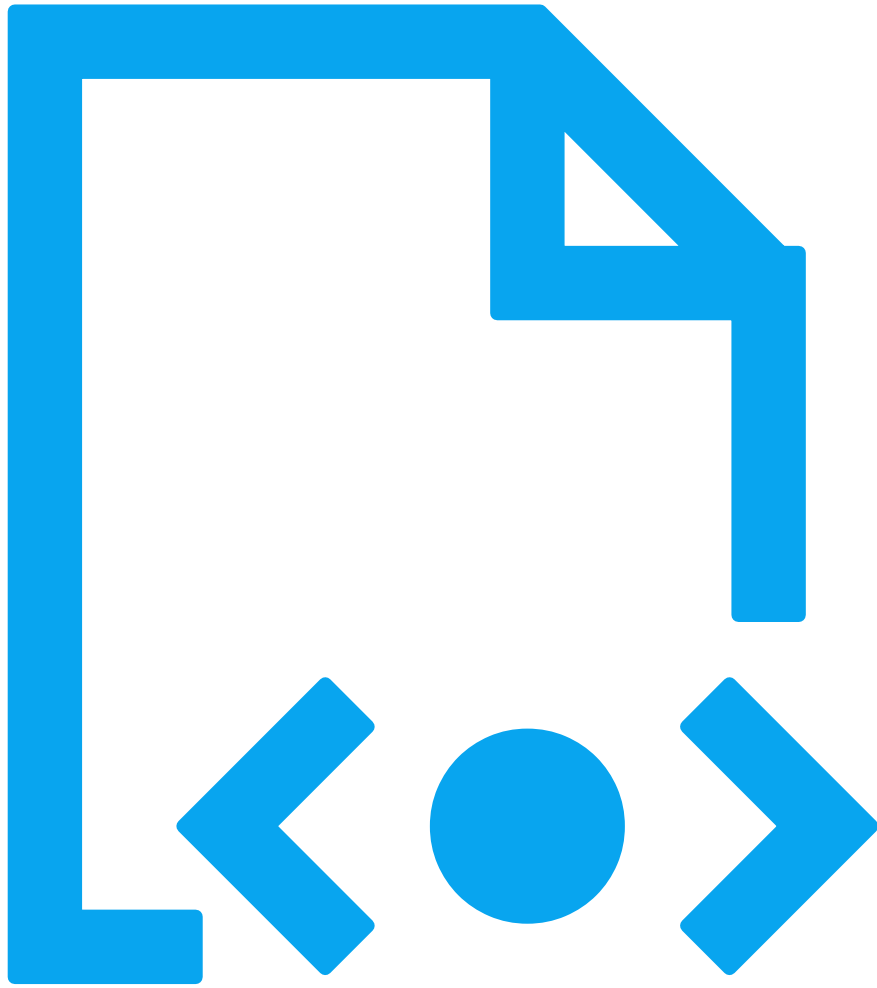
Odd
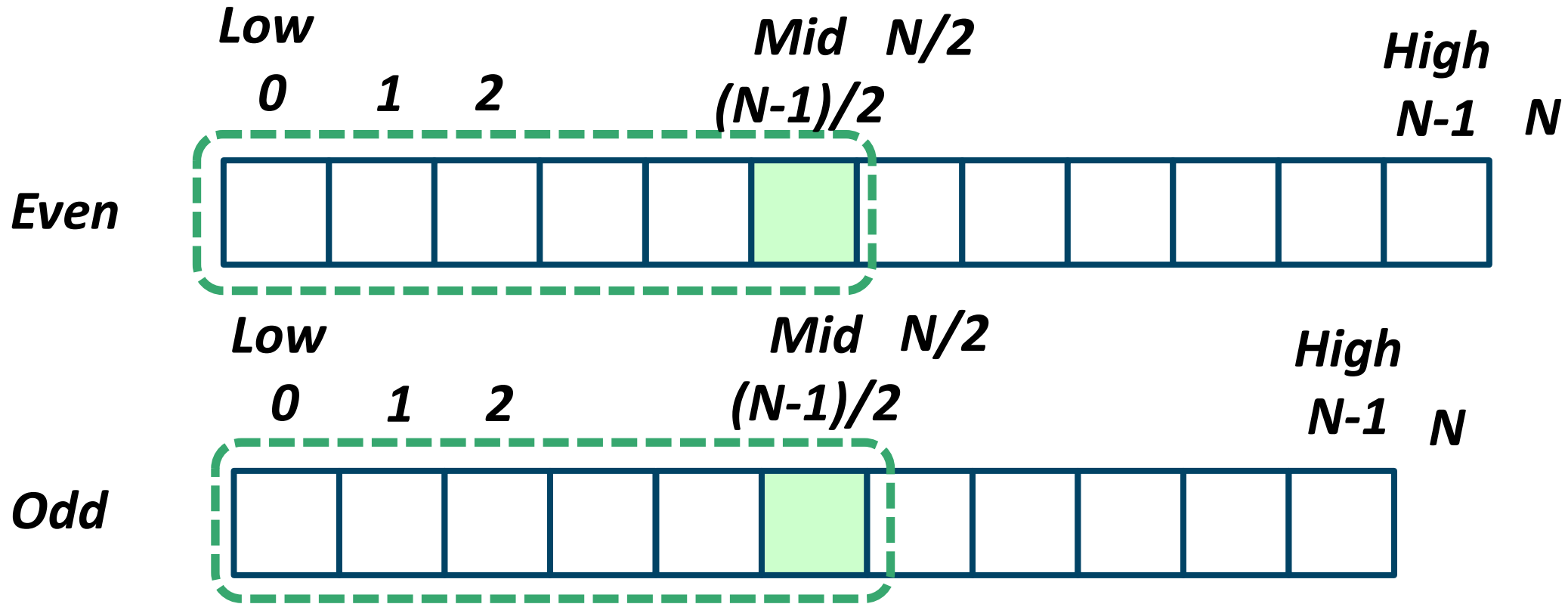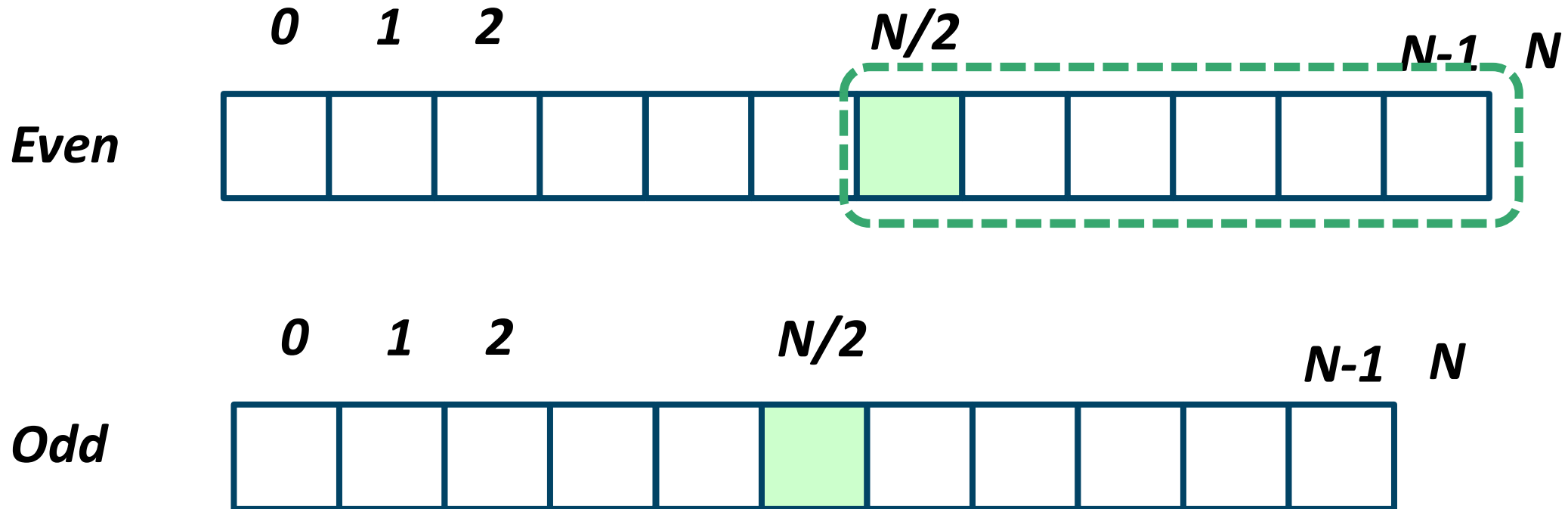
Middle 3
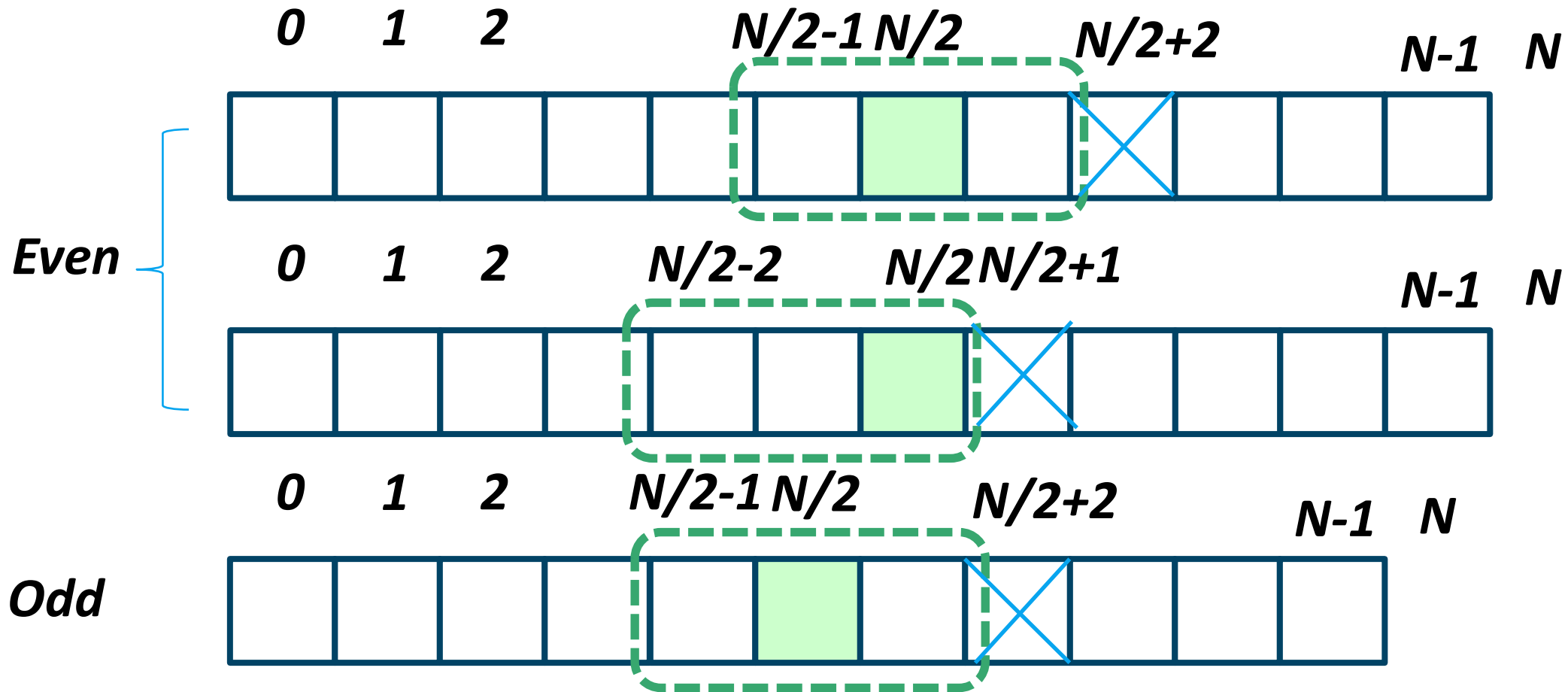
Symmetric Location

Array
Processing

HOW TO PROGRAM?

Array Processing

- Min/Max/Sum
- Largest Index
- Shuffling
- Rotation
- Reverse
- Histogram (Doubloon, canSpell)

Sum

```java
static int[] a = {2, 5, 3, 1, 4};
public static int sum(int[] a){
    int s = 0;
    for (int i=0; i<a.length; i++) s += a[i];
    return s;
}
```

Max 1

```java
public static int max1(int[] a){
    int m = Integer.MIN_VALUE;
    for (int x: a){
        if (x>m) m = x;
    }
    return m;
}
```

Max 2

```java
public static int max2(int[] a){
    int m = a[0];
    for (int i=1; i<a.length; i++){
        if (a[i] > m) m = a[i];
    }
    return m;
}
```

Min 1

```java
public static int min1(int[] a){
    int m = Integer.MAX_VALUE;
    for (int x: a){
        if (x<m) m = x;
    }
    return m;
}
```

Min 2

```java
public static int min2(int[] a){
    int m = a[0];
    for (int i=1; i<a.length; i++){
        if (a[i] < m) m = a[i];
    }
    return m;
}
```

Rotate Left

```java
public static int[] rotateLeft(int[] a){
    int tmp = a[0];
    for (int i=1; i<a.length; i++){
        a[i-1] = a[i];
    }
    a[a.length-1] = tmp;
    return a;
}
```

Rotate Right

```java
public static int[] rotateRight(int[] b){
    int tmp = b[b.length-1];
    for (int i=b.length-2; i>=0; i--){
        b[i+1] = b[i];
    }

    b[0] = tmp;
    return b;
}
```

Reverse 1

```java
public static int[] reverse1(int[] a){
    int[] b = new int[a.length];
    for (int i=0; i<a.length; i++){
        b[a.length-1-i] = a[i];
    }
    return b;
}
```

Reverse 2

```java
public static int[] reverse2(int[] a){
    for (int i=0; i<a.length/2; i++){
        int tmp = a[i];
        a[i] =a[a.length-1-i];
        a[a.length-1-i] = tmp;
    }
    return a;
}
```

```java
public class Histogram
{
    static String s = "abdjslkqiepqruqwoirafdafhdjznxadoauifqprqlsdjfhfhsdgajweyrw";

    public static int[] histogram(String s){
        int[] h = new int[26];
        for (char x : s.toCharArray()){
            h[x-'a']++;
        }
        return h;
    }

    public static void main(String[] args){
        int[] h = histogram(s);
        for (char x = 'a'; x <= 'z'; x++){
            System.out.printf("%c, %d\n", x, h[x-'a']);
        }
    }
}
```