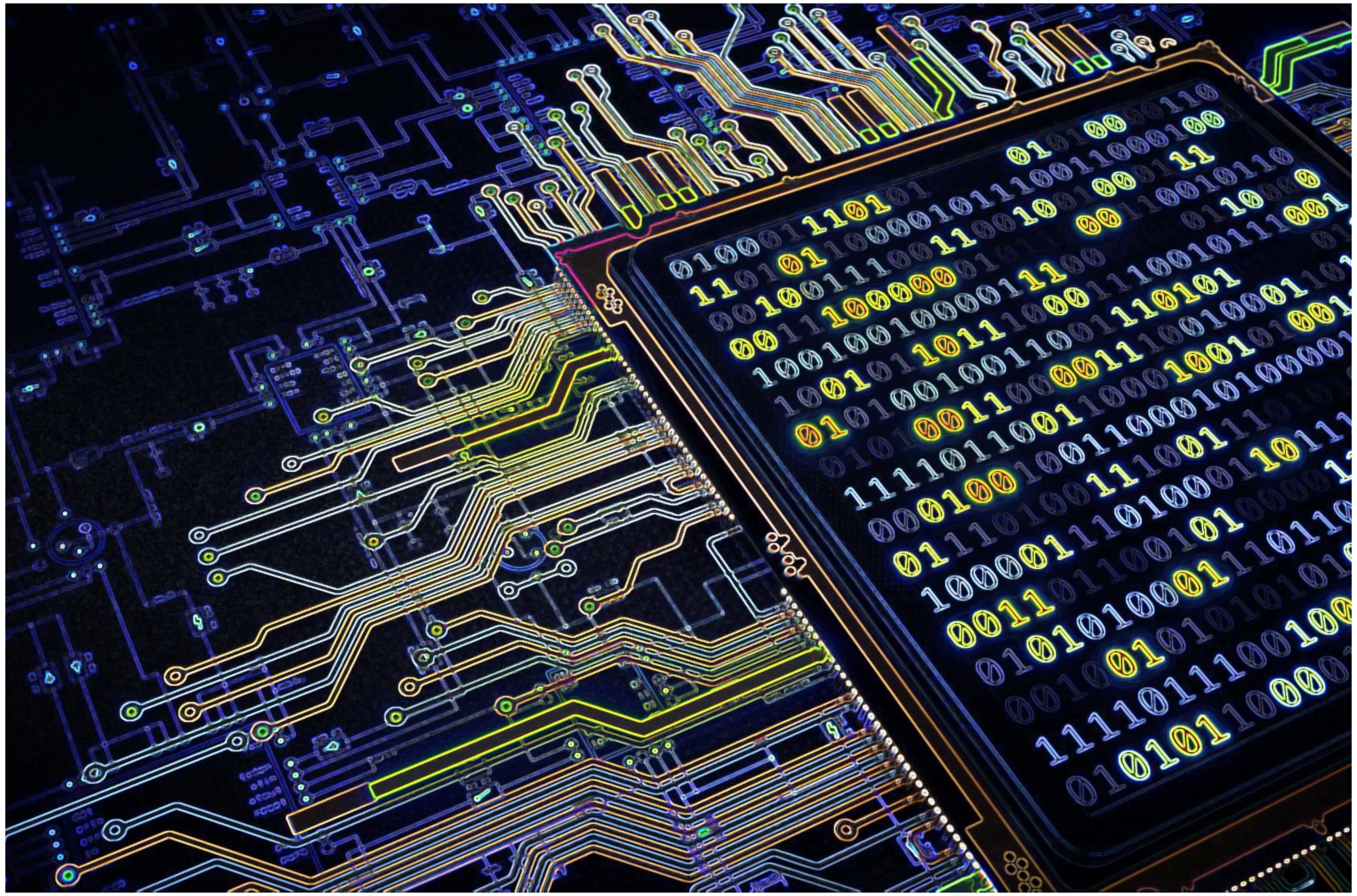


Programming Abstractions

Dr. Ritwik Banerjee
Computer Science
Stony Brook University

The course website is linked from your Brightspace.

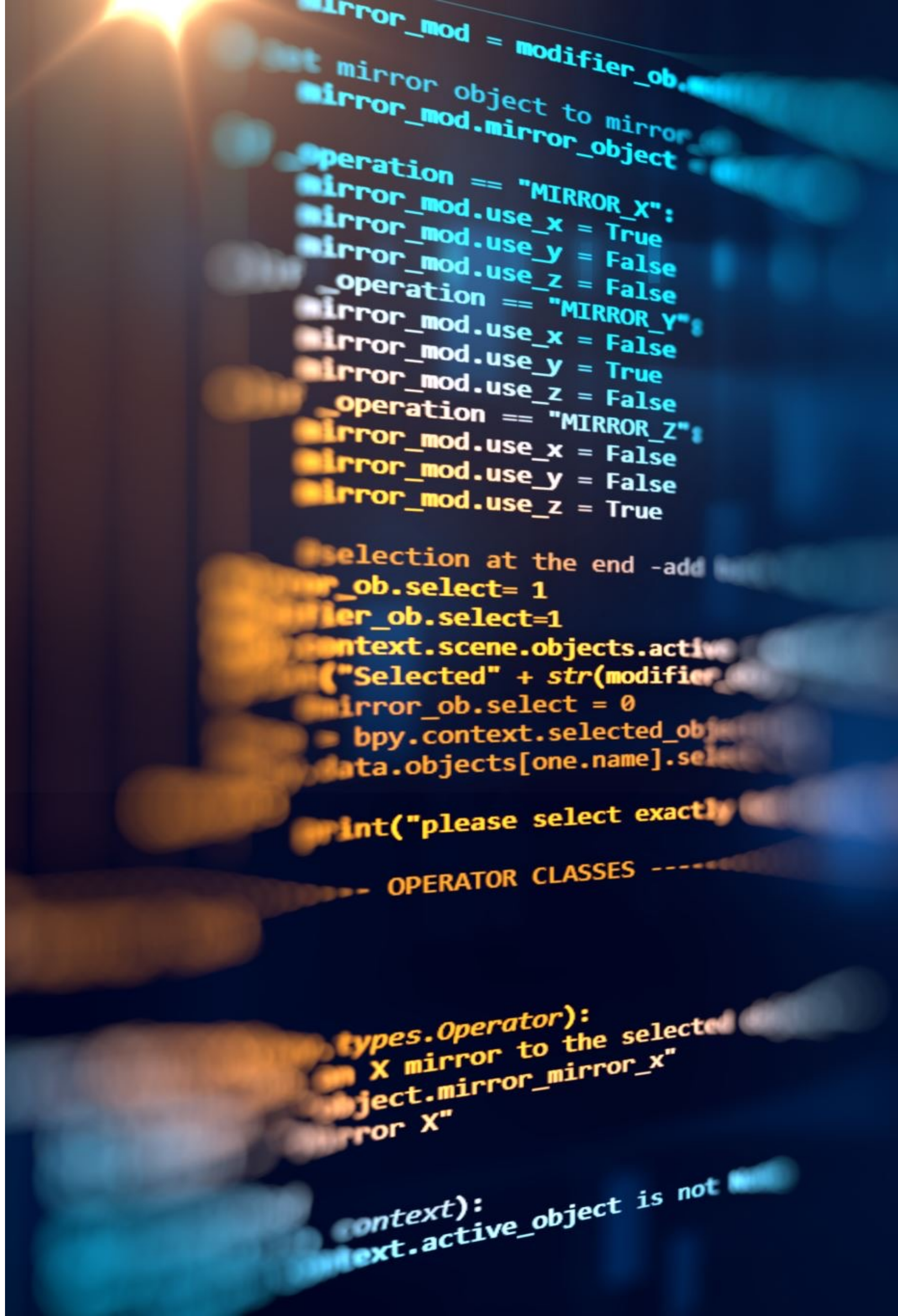
Instructor office hours are held on:

Tuesday, Thursday 6:30 pm - 8:00 pm

(Online; link available from the course website)

How to think of "programming"

- We use three programming languages: OCaml, Java, and Python.
- But the idea is NOT to focus on individual programming languages.
- If you mentally keep thinking in terms of individual languages, you may find this course to be harder than it needs to be!
- Instead, try to think of this course in terms of ideas that go behind a language, and ideas that go into writing **good code**.
- We will start with how programming started, and how people designed different languages based on such ideas. These are categorized into programming paradigms.
- Each paradigm has its own strengths and drawbacks. Our goal is to learn enough about different ways of thinking about programming, so that we know, in a given situation, which language and which paradigm to use.

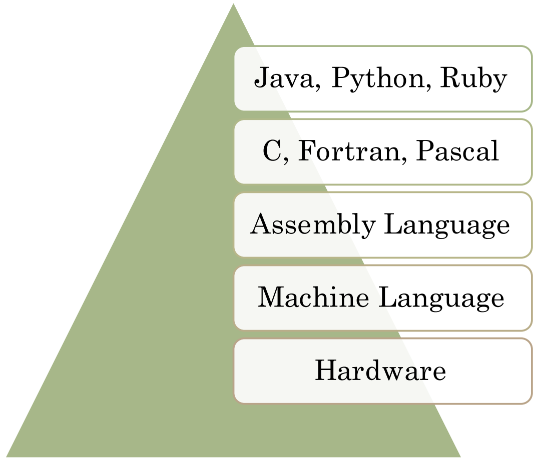


How to think of “abstraction”

- Most programming languages are *high-level* languages, where the phrase “high-level” indicates a higher degree of abstraction.
- The second word of this course’s name – **abstraction** – is among the most important ideas in programming.
- It refers to the degree to which the language’s features are separated away from the details of a particular computer’s architecture and/or implementation at *lower* levels.

We write code to solve problems. So, given a specific problem, writing good code involves

1. using the right **paradigm** for the problem,
2. using the proper amount of **abstraction, and**
3. having adequate modularity in your code.



Course Description

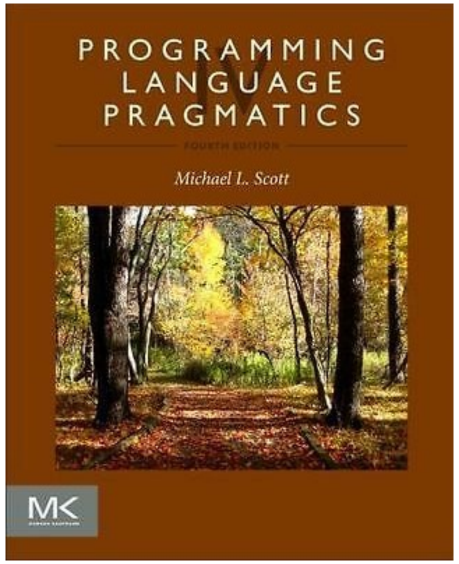
Programming concepts and paradigms, including

- functional programming
- object-orientation
- basics of type systems
- memory management
- program and data abstractions
- parameter passing
- modularity
- software design and development fundamentals
- concurrent programming

Includes weekly recitations

- practice of programming in a variety of high-level languages to gain familiarity with the above concepts

Course Description



Reference book(s) and reading material

- Michael L. Scott. Programming Language Pragmatics.
- For details pertaining to specific programming languages, the recommended material will mostly be from the following:
  - Python tutorial: https://docs.python.org/3/tutorial/
  - The official OCaml learning material from https://ocaml.org/learn/
- Other reading material (if used) will be added to the website for this course.

Prerequisites

1. C or higher in CSE 214
2. CSE major

We will operate under the assumption that you have retained knowledge of

i. data structures and basic algorithms covered in CSE 214,
ii. basic knowledge of Java, as covered in CSE 114 and CSE 214, and
iii. the mathematical and basic logic concepts covered in CSE 215

Specifically, the following are expected:

- the ability to write programs of a few hundred lines of code in the Java programming language,
- an understanding of fundamental data structures, including lists, binary trees, hash tables, and graphs, and the ability to employ these data structures in the form provided by the standard Java API,
- an understanding of the basic principles of recursion,
- the ability to construct simple command-based user interfaces, and to use basic I/O for input
- and output of data, and
- a solid foundation of basic mathematical and geometric reasoning using pre-calculus concepts.

Course Outcomes

An understanding of programming paradigms and tradeoffs.

An understanding of functional techniques to identify, formulate, and solve problems.

An ability to apply techniques of object-oriented programming in the context of software development.

Grading Schema

1. Four assignments: 40% (10% each)
2. Mid-term exam: 20%
3. Final exam: 30%
4. Recitation: 10%

Discussion Forum

- As a technical discussion forum, we will be using the discussion platform on Brightspace. Everyone is expected to
  - use this platform responsibly,
  - always maintain social decorum, and
  - not use this platform for non-technical (especially non-course related) issues.
- Access to the discussion platform is not a required component of the course. As such, any violation of the above decorum (especially if it compromises the social and/or learning environment for other students) by an individual will lead to the immediate removal of that individual from the discussion forum.

- You may discuss the homework assignments and recitation problems in this course with anyone you like.
- **Each submission (including written material and coding), however, must be the student's own work, and *only* their own work**. Any evidence of written homework submissions or source code being copied, shared, or transmitted in any way between students (this includes using source code downloaded from the Internet or written by others in previous semesters!) will be regarded as evidence of academic dishonesty. Additionally, any evidence of sharing of information or using unauthorized information during an examination will also be regarded as evidence of academic dishonesty.
  - *Just FYI, there are some excellent tools like MOSS to compare (n > 1) codes and discover the extent of shared code, even if variable names and other details have been significantly changed.*
- Each student must pursue his or her academic goals honestly and be personally accountable for all submitted work. Representing another person's work as your own is always wrong. Faculty is required to report any suspected instances of academic dishonesty to the Academic Judiciary. Faculty in the Health Sciences Center (School of Health Technology & Management, Nursing, Social Welfare, Dental Medicine) and School of Medicine are required to follow their school-specific procedures. For more comprehensive information on academic integrity, including categories of academic dishonesty please refer to the academic judiciary website at
  - http://www.stonybrook.edu/commcms/academic_integrity/index.html

Academic Integrity

If you have a physical, psychological, medical, or learning disability that may impact your course work, please contact the Student Accessibility Support Center, 128 ECC Building, (631) 632-6748, or at sasc@stonybrook.edu. They will determine with you what accommodations are necessary and appropriate. All information and documentation is confidential. Students who require assistance during emergency evacuation are encouraged to discuss their needs with their professors and the Student Accessibility Support Center.

For procedures and information go to the following website:

- https://ehs.stonybrook.edu/programs/fire-safety/emergency-evacuation/evacuation-guide-people-physical-disabilities

and search *Fire Safety and Evacuation and Disabilities*.

Student Accessibility Support

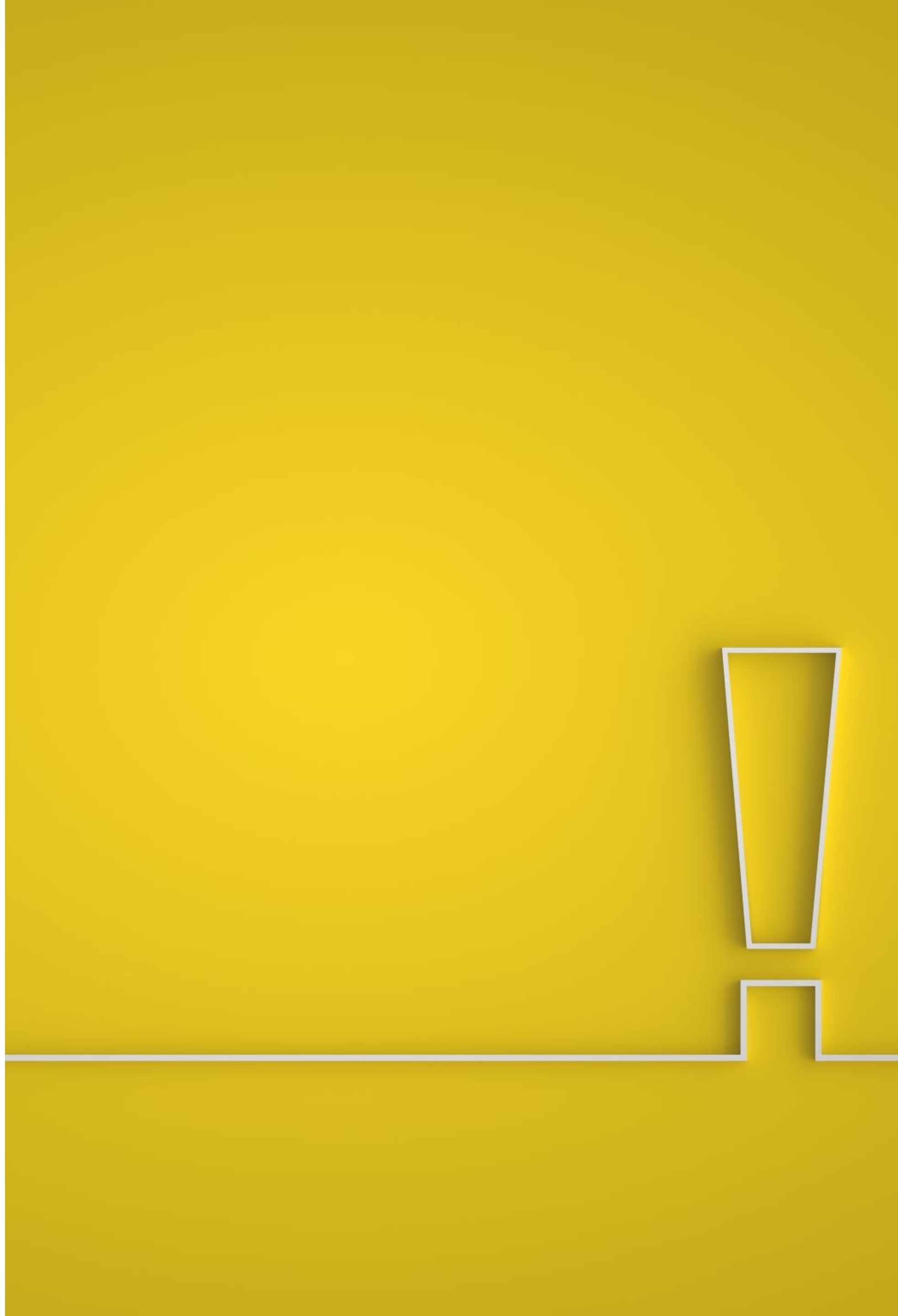
- Stony Brook University expects students to respect the rights, privileges, and property of other people.
- Faculty are required to report to the Office of University Community Standards any disruptive behavior that interrupts their ability to teach, compromises the safety of the learning environment, or inhibits students' ability to learn. Faculty in the HSC Schools and the School of Medicine are required to follow their school-specific procedures. Further information about most academic matters can be found in the Undergraduate Bulletin, the Undergraduate Class Schedule, and the Faculty-Employee Handbook.

Critical Incident Management

Communication Policy

Please follow the communication decorum below as strictly as possible for effective and efficient communication with the instructor and the teaching assistants:

- Do *not* use any email address for the instructor other than rbanerjee@cs.stonybrook.edu.
- Do not use any email address to communicate with a TA other than the ones provided in the web site for this course.
- Clearly state the course and subject matter in your email subject.
  - For example, "CSE 216: homework 3 doubt".
  - If there is an announcement that is sent to the entire class, do NOT hit reply. Send a separate email with a proper subject.
- Follow announcements carefully. **Emails asking for information already available in announcements or on the course site may not receive a response.**
- Even though Brightspace will be used, in-person discussions are encouraged. For this, the instructor or TA office hours are the best venue. Sometimes, we may ask many individual questions sent via email to be posted on the discussion forum, so that a larger audience can benefit and/or contribute.

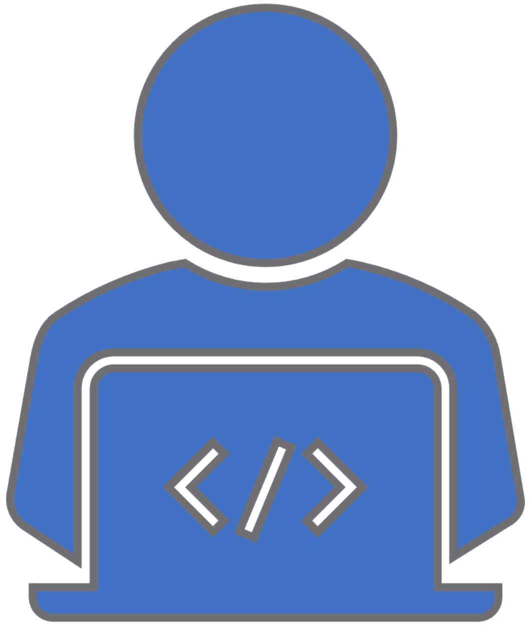


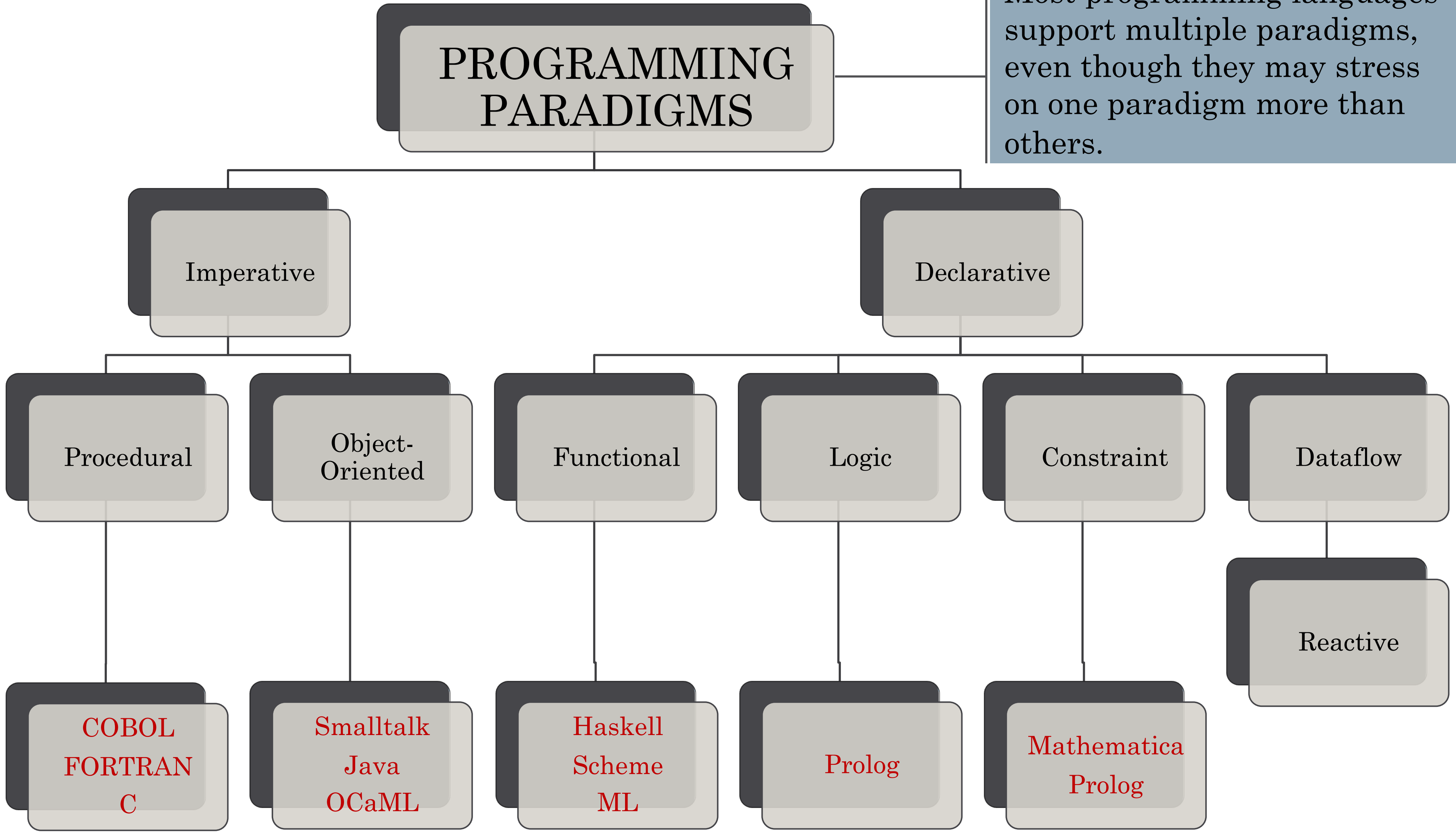
Overview of topics

Programming paradigms	scopes bindings parameter passing type systems
Functional programming	recursion higher-order procedures streams and lazy evaluation
Object-oriented design and programming	abstraction and encapsulation class hierarchy polymorphism and inheritance object-oriented design principles
Cross-cutting concepts	multi-paradigm programming unit testing
Parallel Programming	multithreading coroutines

Programming paradigms

- We can think of a paradigm as a ‘school of thought’ in the world of programming.
- Programming paradigms are a way to classify languages based on their features.
  - It’s a fuzzy classification – a language may belong to multiple paradigms.
  - A particular programming language may make it very easy to write in one paradigm, but not in another.





Most programming languages support multiple paradigms, even though they may stress on one paradigm more than others.

Imperative Programming

Think of imperative programming as a "recipe", where each step is an instruction about how to perform the next action, and the next action depends on the current "state" of your kitchen!

With the exception of *reconfigurable computing* (e.g., FPGAs), the hardware implementation of all computers is **imperative**, i.e., "what" to do is described in terms of "how" to do it.

This is because the hardware is designed to execute machine code, which is written in imperative style.

(Relatively) higher-level imperative languages like C are abstractions of assembly language, but they follow the same paradigm.

Imperative Programming

- A **statement** is a syntactic unit of an imperative programming language that expresses some action to be carried out.
- The program in such a language thus becomes a sequence of statements.

```
assert(x == 7);      /* assertion statement; programmer assumes the
                       value of x to be 7 after this line */
x = 2 + 3;           /* assignment statement */
2 + 3;               /* has no effect; smart compilers will discard
                       this line */
puts("hello world"); /* a function call */
goto label;          /* a goto statement */
return 0;            /* a return statement */

x = sqrt(2); /* an assignment statement containing a function call */
```

Simple Statements

Imperative Programming

- An **assignment statement** performs an operation on information located in memory and stores the results in memory for later use.
  - Higher-level imperative languages permit evaluation of complex expressions that may consist of a combination of arithmetic operations and function evaluations.

```
assert(x == 7);      /* assertion statement; programmer assumes the
                       value of x to be 7 after this line */
x = 2 + 3;           /* assignment statement */
2 + 3;               /* has no effect; smart compilers will discard
                       this line */
puts(“hello world”); /* a function call */
goto label;          /* a goto statement */
return 0;            /* a return statement */

x = sqrt(2); /* an assignment statement containing a function call */
```

Simple Statements

Imperative Programming

- A conditional statement allows a sequence of statements (known as a *block* or a *code block*) to be executed only if some condition is met.
- Otherwise, the statements are skipped, and the execution sequence continues from the statement following them.

```
if (happy) {
    smile();
} else if (sad) {
    frown();
} else {
    stoic();
}
```

```
switch (i % 2) {
    case 0:
        type = EVEN;
        break;
    default:          /* equiv. to case 1 */
        type = ODD;
        break;
}
```

Compound Statements

Imperative Programming

- Looping statements allow a block to be executed multiple times.
- Loops can execute a block a predefined number of times, or they can execute them repeatedly until some condition changes.

```c
while ((c = getchar()) != EOF) {
    putchar(c);
}

do {
    computation(i);
    ++i;
} while (i < 10);

for (i = 1; i < n; i *= 2) {
    printf("%d\n", i);
}
```

Compound Statements

Procedural Programming

A type of imperative programming based on the concept of procedure calls (COBOL, Fortran, C, Pascal).

Procedures (a.k.a. routines, subroutines, or functions), simply contain a series of computational steps to be carried out.

- That is, they define “how” to do what they are being asked to do. They explicitly refer to the underlying **state** (i.e., variables and their values), and are therefore within the ambit of imperative programming.
- Any procedure might be called at any point during a program’s execution. The call may come from other procedures or even itself.

Object Oriented Programming

A paradigm based on the concept of objects, which may contain

- data in the form of **fields**, sometimes called **attributes**, and
- code in the form of **procedures**, a.k.a. **methods**.

An object's procedures can access and often modify the data of the object with which they are associated (using **`this`** or **`self`**).

In OOP, programs are designed by making them out of objects that interact with one another.

- Most OOP languages are class-based, i.e., objects are **instances** of **classes** (usually, this determines their type).

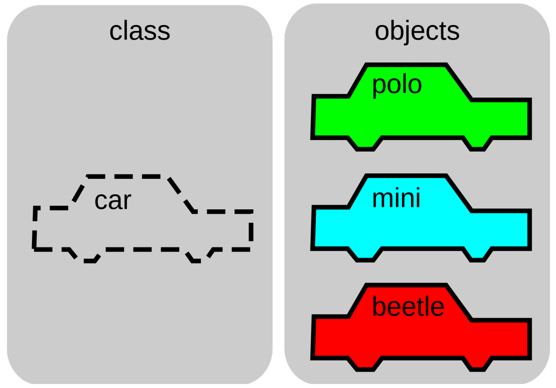
Object Oriented Programming

They have features carried over from older non-OOP languages.

- **Variables** that can store information formatted in a small number of built-in data types like integers and characters.
  - This may include data structures like strings, lists, and hash tables. These structures are either built-in, or result from combining built-in structures using internal pointers.
- Procedures that take input, generate output, and may manipulate data.
- For ease of organization, procedures and variables are usually grouped into files and modules with namespaces.
  - This is called **modular programming**. It is a software design principle, not a programming paradigm.

Objects and Classes

- A class comprises the definitions for the data format and available procedures for a given type of object.
- An object is an instance of a class.



Source: Pluke [CC0], from Wikimedia Commons

- In class-based OOP languages, classes are defined beforehand, and the objects are instantiated based on the classes.
- Given an object, a programmer can safely assume all the properties of the class to which the object belongs.

Object-Oriented Programming

Prototype-based programming

- Also known as *instance-based* or *classless* (as opposed to 'class-based') programming.
- Here, objects are the primary entities. Classes don't even exist!
- Every object has one, and only one, prototype.
  - New objects are created using a pre-existing object as their prototype.
- A programmer may think of two different objects (say, **`apple`** and **`orange`**) as types of **`fruit`** (if the fruit object exists). In this case, both objects have **`fruit`** as their prototype.
  - Thus, the idea of a fruit class is *implicitly* present in the prototype object.
  - The attributes and methods of the prototype are delegated to all objects defined by it.

Object-Oriented Programming

Dispatching and message passing

- It is the responsibility of the object (and *not* any external code) to select the procedure to execute in response to a method call.
    - This is typically done at runtime by looking up the method in a table associated with the object.
    - This is known as **dynamic dispatch**.
    - It is different from abstract data type (ADT) you studied in CSE 214. In ADT, you have a fixed implementation of the operations for all instances.
    - If the choice of the procedure depends on more than one attribute of the object, it is called **multiple dispatch**.
- The method call is also sometimes called **message passing** because people think of it as a message being passed to the object for dispatch.

Object-Oriented Programming

Encapsulation

OOP languages usually provide ways to do the following:

a) restrict access to some of the object's components (attributes and/or methods)

b) bundle the data with the methods operating on that data

- Either (a) or (b) or a combination of the two is known as **encapsulation**.
- Often, (a) just by itself is known as **information hiding**.

```java
public class Person {
  private String name;

  public Person(String name) { this.name = name; }

  public String getName() { return name; }
}
```

What could happen if `name` is not **`private`**?

Object-Oriented Programming

Polymorphism is a concept that refers to the ability of an entity to take on multiple forms.

```java
class Animal {
  public void move() { /* no implementation */ }
}

class Horse extends Animal {
  public void move() { gallop(); }
}

class Bird extends Animal {
  public void move() { fly(); }
}

class AnimalPolymorphism {
  public static void main(String[] args) {
    Animal a1 = new Animal();
    Animal a2 = new Bird();
    Animal a3 = new Horse();

    a1.move();
    a2.move();
    a3.move();
  }
}
```

Object-Oriented Programming

Inheritance is the mechanism of basing an object (or class) upon another object (or class) retaining similar implementation.

- In some OOP languages, subclasses inherit the features of one superclass. This is known as **single inheritance**.
- If one class can have more than one superclass and inherit features from multiple parent classes, it is known as **multiple inheritance**.

Comparison

PROCEDURAL

- Focus is on breaking down a program into a collection of *variables*, *data structures*, and *subroutines*.
- Relies heavily on blocks and **scope**.
- Uses reserved terms like **if**, **while**, **for**, etc. to implement the order in which a program's instructions are executed and evaluated.

OBJECT-ORIENTED

- Focus is on breaking down a program into objects that expose some specific behavior (through *methods*) and data (through *attributes*) using interfaces.

Declarative Programming

In declarative programming, we describe the logic of the program without specifying the order in which a program's instructions are executed and evaluated.

Commonly, we say that declarative programming is about "what" to do, without specifying "how" to do it.

Of course, the computer needs to be told how to do something at *some* point!

But with declarative programming, those details are left to the language's implementation.

There is a *decoupling* of 'what' and 'how', which makes life easier for the developer.

In that sense, they are "higher level" languages.

Declarative Programming

Describes what a computation must do

Database Query Languages. (e.g., SQL – used in relational database management)

Has a clear correspondence with mathematical logic

Logic Programming (e.g., Prolog – used in various AI applications for logical reasoning)

No "side effects"

Functional Programming (e.g., Haskell – used in financial computing, AI, and process automation)

A comparison in the real world

I am right outside the main entrance of the New CS Bldg.
How do I get to your office from here?

Imperative

1. Take the first right turn, walk to the end of the corridor, and take the flight of stairs to the second floor. Then turn left, cross 5 doors on your right. The 6th door is my office.

2. Take the first flight of stairs after room 120 on your left. Exit the stairs on the second floor and … .

Declarative

- My office is Room 206.

Logic and/or constraint-based programming

- Based on predicate logic and an axiomatic way of finding solutions.
- The goal is often to find specific relationships that are true, starting with basic relations that are always true. Such a basic truth is called an **axiom**.
- Perhaps the best-known logic programming language is Prolog.
- These languages are often used to model *constraints*. For example,

find the value of an integer variable **x** *under the constraint that it is different from the value of another integer variable* **y** *and that it must be a strictly positive integer*

Dataflow Programming

- Computation is modeled as a ‘flow of information’ – as a directed graph – between different operations.
- Explicitly defined input and output connect the operations.
  - In that sense, dataflow programming shares some features of functional programming.
  - Each operation can be though of as a ‘black box’ function.
  - A traditional sequential program is like a worker moving around between tasks in a specific order called the **control flow**.
  - A dataflow program is like an assembly line where a worker carries out a task whenever all the inputs are available. Other workers may, in parallel, carry out other tasks if they don’t hinder each other.
    - This paradigm is often very useful in parallel programming.

Reactive Programming

- A declarative, dataflow programming paradigm where it becomes very easy to *propagate* changes in data.
- As an overly simplistic idea, consider a statement such as **`x = y + z`**
  - In traditional imperative programming, the x is assigned the sum of the values of y and z. If the value(s) of y and/or z changes later, the value of x is not affected.
  - In reactive programming, the value of x is automatically updated whenever y and/or z change.
- Reactive programming is extremely useful in interactive applications, and is often used in the model-view-controller (MVC) architecture.

Not within the scope of our syllabus

Functional Programming

1

Based on recursive definitions.

- They are inspired by a computational model called **lambda calculus**, developed by Alonzo Church in the 1930s.

2

A program is viewed as a mathematical function that transforms an input to an output. It is often defined in terms of simpler functions.

- We will see many examples of functional programming in multiple languages (e.g., Java, Python, OCaML).

One Problem. One pseudocode. Three paradigms.

Implementing the greatest common divisor (GCD) solution in different paradigms and different languages.

The pseudocode

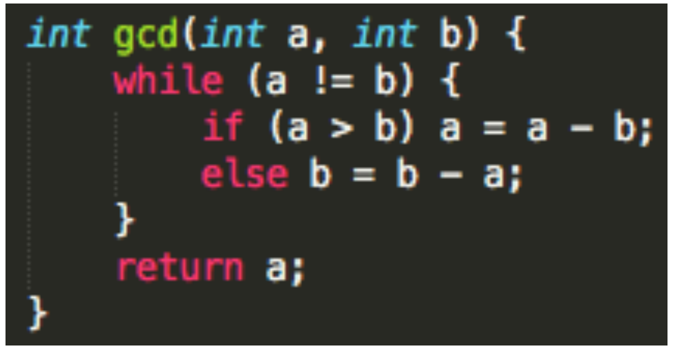
```
function gcd(a, b)
    while a ≠ b
        if a > b
            a := a − b;
        else
            b := b − a;
    return a;
```

Three paradigms

1. **Imperative programming in C**

To compute the gcd of **a** and **b**:

1. check if **a** and **b** are equal
2. if they are, then print **a** and stop
3. otherwise, replace the larger number by their difference and repeat steps 1 and 2



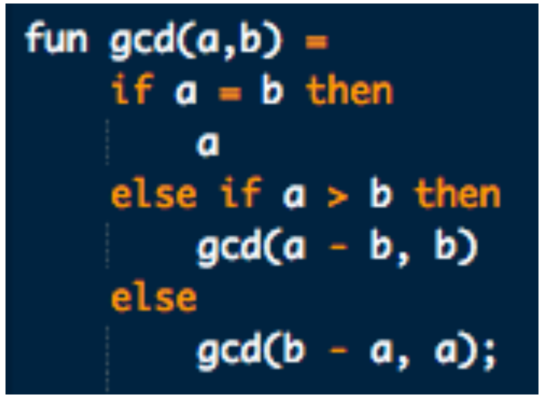
```c
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a = a - b;
        else b = b - a;
    }
    return a;
}
```

Three paradigms

2. **Functional programming in ML**

To compute the gcd of **a** and **b**:

1. check if **a** and **b** are equal
2. if they are, then print **a** and stop
3. otherwise, replace **a** by the absolute value of their difference and recursively compute the gcd of this new pair



```
fun gcd(a,b) =
    if a = b then
        a
    else if a > b then
        gcd(a - b, b)
    else
        gcd(b - a, a);
```

Three paradigms

3. **Logic programming in Prolog**

The proposition **`gcd(X,Y,G)`** is true if

1) **`X`**, **`Y`**, and **`G`** are all equal, or

2) **`X > Y`** and there exists a number **`c`** such that **`c`** is **`X-Y`** and **`gcd(X,Y,G)`** is true, or

3) **`Y > X`** and **`gcd(Y,X,G)`** is true.

To compute the greatest common divisor of **`X`** and **`Y`**, search for a number **`G`** for which the above rules prove that **`gcd(X,Y,G)`** is true.

```
gcd(X,X,X).
gcd(X,Y,G) :- X>Y, Y1 is X-Y, gcd(Y,Y1,G).
gcd(X,Y,G) :- X<Y ,gcd(Y,X,G).
```