OOP Overview, Computer Architecture, Agile Development

In this course, we will be learning many object oriented languages. But, what does this term refer to?

Object Oriented vs Procedural Programming

In a purely procedural style, data tends to be highly decoupled from the functions that operate on it.

In an object oriented style, data tends to carry with it a collection of functions.

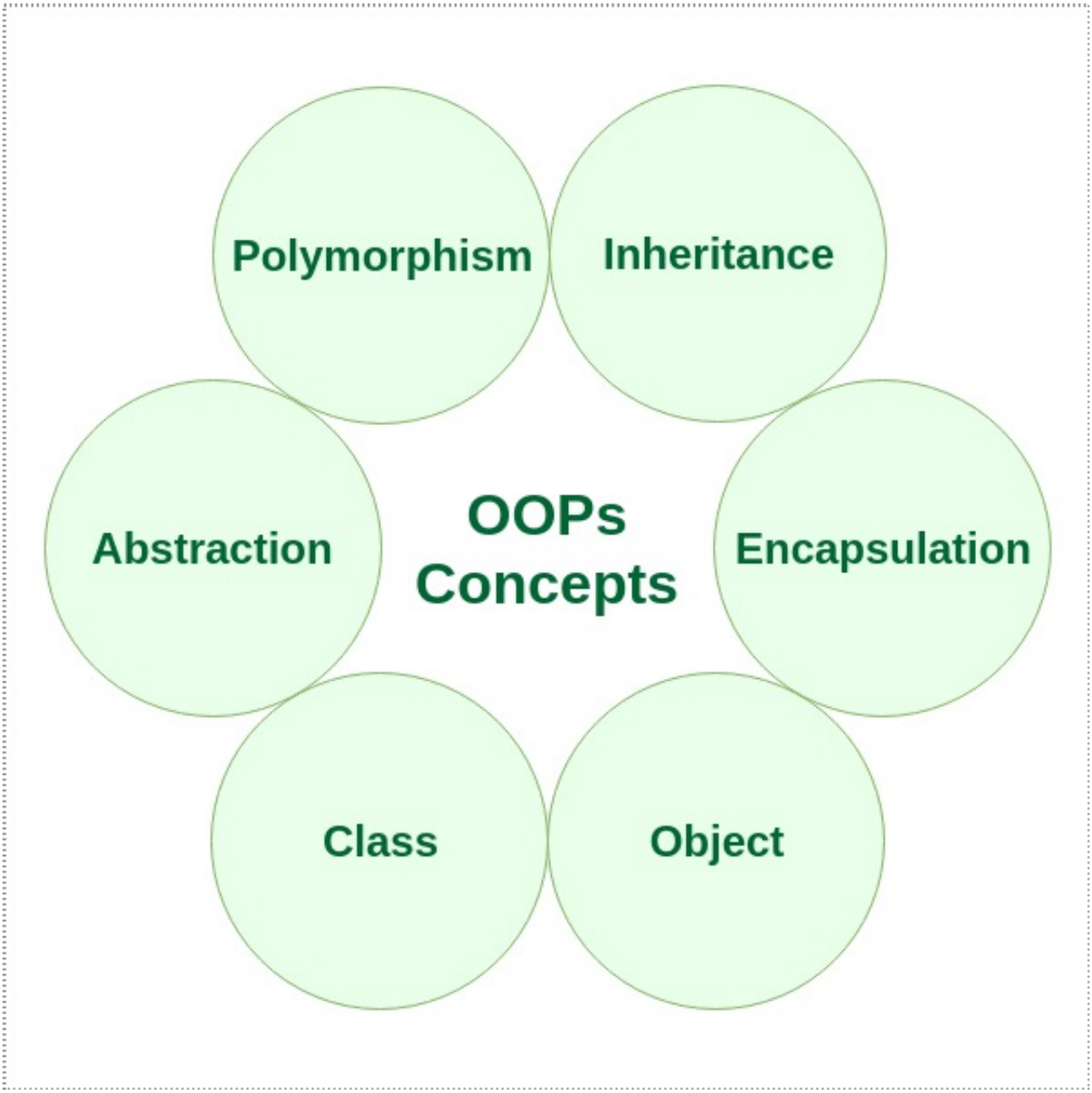
In a functional style, algorithms tend also to be defined in terms of recursion and composition rather than loops and iteration.

helpful definitions:

algorithm - a well-defined procedure that allows a computer to solve a problem.

recursion - method of solving a problem where the solution depends on solutions to smaller instances of the same problem.

advantages of object oriented langauges:



Object-oriented languages are good when you have a fixed set of operations on things, and as your code evolves, you primarily add new things. This can be accomplished by adding new classes which implement existing methods, and the existing classes are left alone.

Functional languages are good when you have a fixed set of things, and as your code evolves, you primarily add new operations on existing things. This can be accomplished by adding new functions which compute with existing data types, and the existing functions are left alone.

When evolution goes the wrong way, you have problems:

Adding a new operation to an object-oriented program may require editing many class definitions to add a new method.

Adding a new kind of thing to a functional program may require editing many function definitions to add a new case.

Modern programming languages like Java, C# etc. follow the Object Oriented approach. In object oriented programming, importance is given to data rather than just writing instructions to complete a task. An object is a thing or idea that you want to model in your program. An object can be anything, example, employee, bank account, car etc.

Class, Object.. what's that?

For getting started with object oriented programming we would have to know what is a class and object and the difference between them. A class is a blueprint for creating an object. It is similar to the blue print of a house.

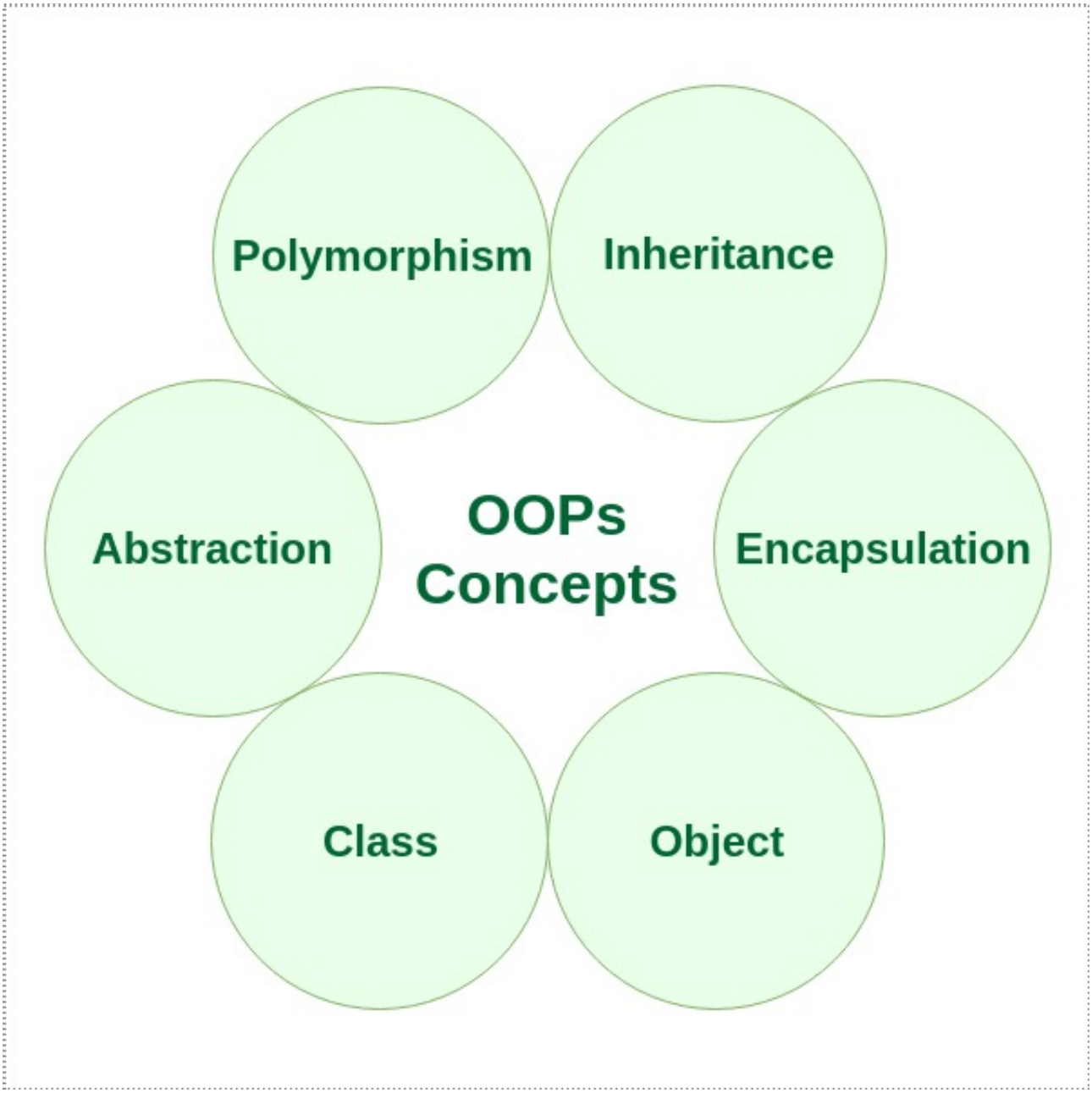
Illustration of Class and Object

A class defines attributes and behavior. If we are to model a car in our application then we could define attributes of the car like, model, fuel, make and behaviors like start, break, accelerate etc.. If you notice, the attributes and behavior that we are specifying are not specific to just one model of car. We are trying to generalize a car by stating that the car which we are going to model in our program will have these number of attributes and behavior.There might be others as well but our scope and interest for the business requirement are limited to these attributes. This helps us create a blue print of the car and later while we use this class for creating objects we create car objects with specific details.

Example, using the same class "Car" we can create different objects having variation in model, fuel type and make year while having the same common behavior.

Object 1 Model Volkswagen Polo Fuel Petrol Make 2017 Start() Break() Accelerate()

Object 2 Model Volkswagen Vento Fuel Diesel Make 2017 Start() Break() Accelerate() In this way, Object oriented programming allows you to easily model real world complex system behavior. With OOP, data and functions (attributes and methods) are bundled together within the object. This prevents the need for any shared or global data with OOP, which is a core difference between the object oriented and procedural approaches.



Abstraction

Abstraction lets you focus on what the object does instead of how it is done. The idea behind abstraction is knowing a thing on a high level. Abstraction helps in building independent modules which can interact with each other by some means. Independent modules also ease maintenance.

Abstraction means to represent the essential feature without detailing the background implementation or internal working detail.

We try to selectively focus on only those things that matter to us or in the case of programming, to our module. Modifying one independent module does not impact the other modules. The only knowledge one needs to know is what a module gives you. The person who uses that module does not need to bother about how the task is achieved or what exactly is happening in the background.

Abstraction is everywhere. Everyday objects that we use has abstractions applied at various levels. One example is applying breaks in your car or bike. The breaking system is abstracted and you are provided with a paddle for stopping your vehicle. Making changes to acceleration system does not affect the braking system they are independent. You also do not have to bother about the internal working of the brakes, you only have to press the brake pedal and be it disc brake or drum brake, the vehicle stops.

Encapsulation The second concept Encapsulation is closely related to Abstraction. Encapsulation is all about exposing a solution to a problem without requiring the consumer to fully understand the problem domain.Encapsulation is binding the data and behaviors together in a single unit. This prevents the client or the user of the module from knowing about the inside view where the behavior of the abstraction is implemented. The data is not accessed directly. It is accessed through the exposed functions. Hiding the internalsÂ of the objectÂ protects its integrityÂ by preventing users from setting the internal data of the component into an invalid or inconsistent state.

Encapsulation isÂ not just about hiding complexity but conversely exposing complexity in a fail-safe manner.

Inheritance Inheritance is a powerful feature of Object oriented programming languages.Inheritance helps in organizing classes into a hierarchy and enabling these classes to inherit attributes and behavior from classes above in the hierarchy. Inheritance describes an “IS A relationship. This is how we talk in the real world. Example. A Parrot is a bird. US Dollar is a type of currency. But the phrase, Bank is a bank account is not correct. This relation is obvious when you try to describe some entity in the given business/problem statement.

With inheritance, you can define a common implementation/behavior and later for specialized classes you can alter or change it to something specialized. Inheritance does not work backward. The parent won’t have properties of the derived class.

Inheritance is a mechanism for code reuse and can help in reducing duplication of code.

Do not force inheritance. You would just be writing unnecessary code. It is important to note that while trying to model the requirement don’t go adding multiple levels of inheritance. This is not needed. You need to try to identify common attributes and behavior in the entities you modeled and based on that you can go ahead re-factoring the code defining a suitable parent class. Common implementation can then be moved to this class.

Polymorphism Polymorphism is the concept that there can be many different implementations of an executable unit and the difference happen all behind the scene without the caller awareness. PolymorphismÂ allows computer systems to be extended with new specialized objects being created while allowing current part of the system to interact with a new object without concern for specific properties of the new objects.

For example, if you needed to write a message on a piece of paper, you could use a pen, pencil, marker or even a crayon. You only require that the item you use can fit in your hand and can make a mark when pressed against the paper. So the action of writing would help you make a mark on the paper and what marking or writing instrument to use is a matter of decision. Another example is a

plane and space shuttle both can be termed as Flying objects. But the way both fly are different I.e. there is a difference in implementation. But from a viewer's perspective both the objects can fly.

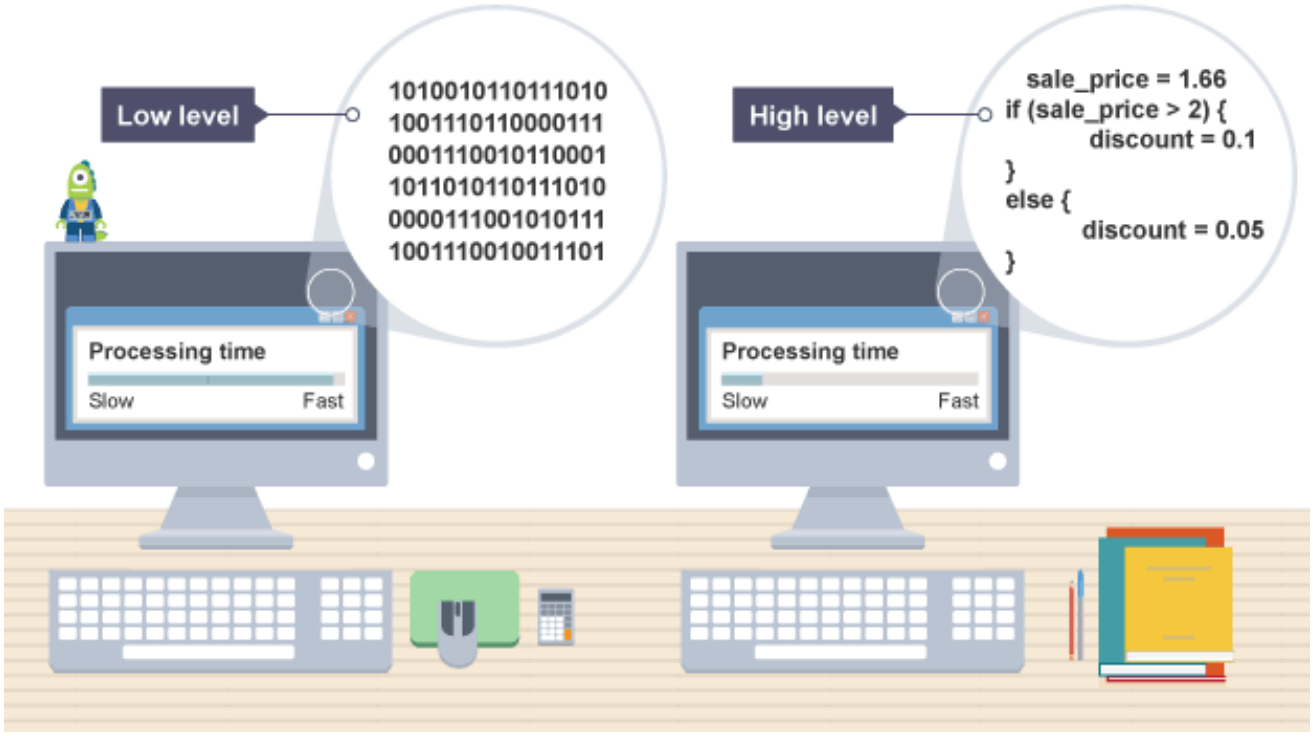
Inheritance is one means of achieving polymorphism where behavior defined in the inherited class can be overridden by writing a custom implementation of the method. This is called method overriding also known as Run Time polymorphism.

There is also one more form of polymorphism called method overloading where inheritance does not come into the picture. The method name is same but the arguments in the method differ.

Computer Architecture

High Level and Low Level Languages

There are two main types of programming languages, high level and low level. Their level depends on how similar the language is to the only language a computer understands, machine code (I'll talk more about machine code soon). High level programming languages are ones most developers are familiar with: Java, Ruby, C++, Javascript, etc. A high level programming language is a language that allows you to tell a computer to do something, but in a syntax that is easy and intuitive for you to understand. It is a totally different language from what a computer understands. In contrast, a low level programming language is only a slight derivation from machine code (if any derivation at all). It's much less human readable, yet easier and faster for the computer to understand. It also takes much less memory than a high level language. The most common low level languages are assembly languages. They group variations of symbols and letters to represent different aspects of the machine code. They use an assembler to convert the assembly language to machine code. Think of the assembler as a dictionary. In a dictionary, you can look up a word and find the meaning. An assembler can "look up" assembly codes and return the "meaning" or machine code. Now, say for instance machine code is the English language, and it's the only language a computer can understand. A high level language is like speaking to the computer in Mandarin, while a low level language is like speaking to it in Pig Latin. It is much easier for the computer to understand low level Pig Latin, which is very similar to English, than it is for it to understand high level Mandarin. Regardless, a computer only understands machine code (or "English"), and needs any language that's not machine code to be either complied into machine code or interpreted (translated).



Complied vs Interpreted Languages

While low level assembly languages are understood by converting the language to machine code using an assembler, most high level languages are understood by using either a complier or interpreter. Let's discuss interpreted languages first since they have a different process than complied or assembly languages. Interpreted languages are not converted to machine code. Instead, any program (or source code) written in these languages are directly read line by line by an interpreter. An interpreter is a computer program that executes the actions in the source code in a similar way that a computer can execute machine code. Using an interpreted language allows for faster coding and testing because it can run and give results immediately, unlike compiled languages. What's also good about these languages is that it can be used on multiple platforms and operating systems since it is not converted to machine code, which is often computer-specific. A disadvantage is that, since it always interprets the code as the code is executed, it can make it much slower than using a compiled language. Python, Perl, and Ruby are popular examples of interpreted languages.

Then there are compiled languages. Compiled languages have to go through a compiler before they are executed. The compiler converts the program into machine code so that it can be understood by the computer. Java, C++, and C# are popular compiled languages. The benefit of compiled languages are that once compiled, they tend to run much faster than interpreted languages. Compiling only has to be done once, then a program can run many times. Interpreted languages on the other hand are consistently being read and can therefore be slower. Another benefit is that, since it is complied, it doesn't reveal any source code, which means people can't see or copy your code. A downfall is that they usually don't work across platforms. A program that runs one way on a Mac doesn't run the same way on a PC. This is why when you go to download applications on the internet, they have an option to download for Mac OS X or Windows.

From the Complier, to Machine Code, to the Processor

After the compiler (or assembler) converts the high level language to machine code, the computer is ready to execute actions. To understand how the processor of the computer (also known as the Central Processing Unit or CPU) reads machine code, you first need to understand what machine code is.

Machine code is a set of binary instructions consisting of 1's and 0's called bits. To the processor, 1 represents an electrical switch being on, while 0 means a switch is off. The 1's and 0's are grouped together in different ways, creating 8-bit combinations called bytes. Combinations of these 1's and 0's send various electrical signals to the transistors in the CPU. Modern CPU's have over a billion transistors which contain logic gates. The logic gates are opened and closed based on the bytes of code it receives, and this opening and closing of the gates is what allows your computer to do what you tell it to do.

Binary	01001000	01100101	01101100	01101100	01101111	00100001
TEXT	H	E	L	L	O	!

In summary, there many different things to consider when choosing a programming language. If you are developing a program that will only be used on one platform or operating system, it may make more sense to use a compiled language. If you are developing a program to be used on multiple platforms, an interpreted language may be your best bet. Nonetheless, programming languages evolve very rapidly, and one language may soon be better suited than the other regardless of how many platforms are used.

Agile Devlopment

Welcome to Agile workflow. This is a word you've probably heard before; it's a hot topic in technology. Other words surrounding Agile include: scrum, scrum master, burndown charts, user stories, acceptance criteria, points, and continuous delivery. Don't worry - we'll go through all of these later.

A Brief History / The Problem

When software development and programming became a widespread industry - we didn't know what we were doing, and we were doing it poorly. Before Agile came about, **Waterfall** was an industry standard delievery method, which in conjunction with the fast paced tech boom in the 1990s, led to some serious **application delivery lag**.

- Application Delivery Lag: Time between buisness needs being validated and production-ready software was up to 3 years. This was super frustrating.
- Waterfall Delivery Method: A very sequential way of software development. Only after the entire

project was done and ready, would it be deployed. Originated in the construction and manufacturing industries - not much changed when development started using it.

How Agile Attempts to Fix It

Agile aims to address the main issue with planning: humans suck at planning. This is not to say that using Agile will make us any better at planning. Agile functions in a way which *allows* for changes to the plan. It *allows* for changes in the development pattern through a short, repetitive, developement-feedback cycle of **sprints**, wherein the developers chip away at the project as a whole by completing **tasks** written with **user stories**.

- Sprint: Typically a 2 week period of development. Sprints are planning, user stories are assigned, and by the end of the sprint, completed stories are typically pushed to.
- Task: A single deliverable item of development
- User Story: A "task". This is just a piece of the puzzle. It describes a small part of the application as seen or experienced by the user.

The Agile Heirarchy for any Project

It is important to remember that these hierarchies are not absolute, nor are they law for practicing agile methodologies. Some companies will want to do it a certain way, and some companies will find that a slightly different process works for them.

1. Sub-Task
    1. This is the smallest item in Agile practice
    2. The completion of multiple sub-tasks should mark the completion of a task. Though, it's not always necessary to break tasks up in sub-tasks
2. Task
    1. A task is a single development effort. And example task could be
        1. "Make all images in our gallery page be square instead of native ratio"
        2. "Build product info component to be reused across the site"
    2. Well defined tasks should have acceptance criteria, a description (*user story*) and a point value assigned.
3. Epic
    1. This is a collection of tasks which all bring an area of the project closer to fruition
    2. Epics typically are large scale initiatives in the project which could go on for many sprints
    3. For the online presence of Target, you may find epics such as
        1. "1 Click Checkout"
        2. Ai Product Referrals System

and so any story that helps bring their 1 Click Checkout product closer to launch would be filed under that epic.

4. Themes / Initiatives
    1. Themes and inititives are essentially the same as Epics, but on a larger scale, and are not always needed or used.

Writing User Stories

User Stories are widely used throughout agile methodologies and it is important that we understand how to read and (sometimes) write them. I say sometimes because in a full fledged agile environment, you'll *potentially* have a scrum master, project manager, BA, etc. writing user stories for you.

Let's start with the basic layout of a user story:

*As a **type of user**, I want **some feature** so that **some reason**.*

Let's break that down into it's 3 parts:

1. Type of User: This is the end user for this effort. Who will this task ***directly*** benefit?
2. Some Feature: This is the thing. The thing that your development effort should accomplish.
3. Some Reason: This is the why. Why is this feature important? What's the reason for someone to want this?

Here are some example user stories:

- As an Amazon Prime subscriber, I want 1 Click Shipping to be disabled by default so that I don't accidentally purchase items I don't need.
- As a Chipotle cashier clerk, I want a list of common orders so that I can get customers through the checkout process quicker.

So as you see - tasks and user stories aren't *just* meant for the customers of our business. They're meant for *any* development effort. Common tips to remember when writing user stories are:

1. It's a user **story** not a user **novel**. Keep it short
2. Only write about 1 piece of functionality. Keep them small in this fasion
3. Always write from the perspective of the user. It'll help keep the important things important

Story Sizing

Learning to use the syntax for appropriate user stories is only half the battle. Arguably, the more important aspect of a good user story is it's size, or **scope**.

It's very possible to get carried away in writing a user story which describes one LARGE piece of functionality. It's still just one piece so it seems fine, however these large stories become harder to estimate, track, and deliver. Afterall - the one of the main goals of Agile and Scrum is to reduce the size of deliverables and limit the current 'work in progress'.

We'll touch back here after reviewing estimation.

IN CLASS ACTIVITY:

Before we go any further - it's time for an activity. Break people off into groups. Descibe a robust feature to be added to an online application - but describe it as a user story (I've described one below - feel free to use). This should be intentionally large (like something worth a lot of story points). Then prompt the class:

1. How can we break this story or task down into smaller deliverables so that we can reach the same goal *iteravely* - that is to deliver small parts at a time, instead of just everything at once.
2. Write those individual tasks down.

We have an ecommerce website with products and an online shopping cart, but no search bar functionality on the site. As a user, I want to start typing in a search bar and see possible options based on my current search query (name, descriptions, meta data, restricted returns). Then as my query gets more specific, my options get more specific.

My answer

1. As a developer, I want to reach an API endpoint with a search query and get back matching results so I can implement a search bar.
2. As a user, I want to input a query in a search bar, and get taken to a page of results based on product name so I can search for products.
3. As a user, I want my query to also consider product descriptions so I can be more vague with my search
4. As a user, I want my possible products list to include results from meta-data as well
5. As a user, I don't want to see results of products that are currently out of stock, so I only get back results that I can purchase

Then have teams read out their revised lists of user stories. Call out any stories that still seem too big and prompt how we could break those down even further. Take it far. Get nit-picky. The main goal here is to take a user story which seemingly has 1 function (in this case - a search function) and turn that into many smaller stories - each with a specific bit of functionality to be added to the previous. This is to get people to think about the MVP of this story and then build and iterate on top of that.

Acceptance Criteria

User stories are great because they tell us what needs done. However, this isn't very conducive to making good software if we don't know how to test it. Acceptance Criteria (AC) defines clear pass/fail tests to run. Development on this story is ready to move forward once all the acceptance criteria statements can be said or tested with a positive result.

AC should describe **intent**, and final functionality, but not necessarily what the **solution** is. For example, the Intent statement below is highly preferred to the Solution statement. Leave the problem solving for the devloper to whom this task is assigned.

- **Intent** - Filter dropdown list should load all options on page load
- **Solution** - Use API call to get list of filter options for drop down on page load

To boil it down - good AC should leave no questions as to how to test this new software, and it should sit alongside the user story to help make the task more robust, detailed, and easier to work with.

Estimating Tasks

Estimating tasks in Agile goes against most estimation tactics we may have used or heard of before, and for good reason. Agile estimation is typically done by assigning **story points** to each story.

- Story Points - This number repesents the relative *difficulty* of the task at hand.

Understanding just what story points represent is very important in proper estimation. *Story points **do not** represent how long a given task will take (e.g. 8 pts is not 8 hours, or 1 day).* We also do not represent estimations of tasks by providing due dates (e.g. this task will be done by Jan 23rd).

There are also some guidlines about which numbers we should use when giving estimates. Typically, we want to use numbers from the Fibonacci sequence (1, 2, 3, 5, 8, 13, 21, 34, 55, 89, etc.). This may seem odd - but there is some psychology at work here and it essentially just make us be a little more *mindful and thoughtful* about our estimations. Notice that did not say "*more accurate*" at estimations. Only practice, time, and repetition, can help us get a little better at estimating.

To touch back on Story Sizing for a minute, let's consider some high rated tasks. If you have a task with a 21pt rating, the first question you should ask yourself isn't *"How am I going to tackle this large story?"*. Contrarily, your first question should be *"How can I break this story down into smaller, more manageble tasks?"*.

Instead of having a 21pt task, It's better (and more Agile) have 5, 8, 8 point stories. You broke that 21pt task into 3 tasks. Can you separate one of those 8pt tasks into 2 smaller ones so that you have 3, 5, 5, 8 point tasks?

Sprint Planning

The sprint planning meeting is pretty self explanatory in terms of what the outcome is: A list of all the tasks that our team will attempt to deliver during the course of this sprint. This meeting should happen 1-2 days before the end of the current sprint, so that any tasks we know for sure won't get done currently, can be added to the coming sprint as well.

This will vary from team to team, depending on what works for you:

- This could be where the tasks get assigned to someone.
- It could also be the time and place for assigning story points (if they weren't assigned at the time of making the task).
- If there are some large stories (21 points or more) - then this could be the time to talk about that and try to break it into smaller tasks.
- If AC isn't written yet, make sure your tester/QA is in the room and start writing some AC for those tickets!
- Was there any take away action items from your most recent Retrospective meeting that could be taken during this planning meeting?

Forecasting

— stretch: todo —

Retros / Reflections

Ah. Reflections. Time to sip some tea and reflect back on all the work you and your team did these past 2 weeks (or however long your sprint was)! This meeting should happen at the end of *every* sprint and it's goal is to figure out what went well this past sprint, what didn't go well, and what would change for the next one. This is a huge component for Agile. Since Agile is all about reducing the size of deliverables and shortening the cylce between work and review, it would make sense that we should even review the process itself.

This meeting is pretty easy to run, but can be awkward at first until your team really learns to trust each other and value each other's opinions. Best done with a white board and some post it notes - here's how it works.

1. Divide a whiteboard into 2 sections: **What went well** and **What didn't go well**
2. Allow about 5 minutes for everyone to fill out as many post it notes as that can think of and put them in their respective category
3. Then go through all of the post its and let people explain their positions and opinions.
4. After that, have an open discussion to figure out a couple of action items to specifically keep in mind and work on during your next sprint (communication, better AC, etc.)

Another useful reflection tool used are Burndown Charts. They're also used to track current progress during the sprint, but they tell a pretty great story at the end. Let's look at a few charts and see what they say about a 2 week, 10 day, sprint.

[IMAGE 1] - The Par Sprint

The line here that shows our "actual" points remaining stays pretty close to the burndown line throughout the sprint, and it ends the sprint off by getting our last task through testing. This is a solid sprint and our team should feel great about that.

[IMAGE 2] - The Neverending Sprint

Here, we see our "actual" points line actually *increase* twice during this sprint. Whenever this happens, 1 of 2 things have occurred:

1. We've added in new tasks in the middle of the sprint, so their points get added to our current scope
2. We change some of our tasks to have a higher story points estimation

Both things aren't great. Sometimes something comes up and we *need* to get some **hot-fixes** through right away, and that's perfectly fine. We shouldn't ignore serious flaws in our technology just because some chart is telling us we don't have time. However, we should document that. If this type of thing tends to happen often - then it may be useful to always include a 5 or 8 pt story in our sprint and set it aside for these mid-sprint asks. Our team shouldn't feel bad for not getting everything done, but we should have some discussions in our Retrospective meeting about what happened and how to correct.

[IMAGE 3] - The Eager Sprint

The phrases "Your eyes were bigger than your stomach" or "You bit off more than you could chew" comes to mind here. This type of burndown chart, where your current progress line doesn't peak, but also doesn't keep up with the burndown line, and ends with a positive "points to go" value, indicates 1 of 2 things typically:

1. Our estimations were generally too low. If we took on a bunch of work that turned out to be more difficult than we thought, it makes sense that we would fall behind.
2. Our estimations were correct - but we added too many tasks to this sprint. That's fine! How many points did you have left?? When you plan your next sprint, try to include that many fewer points as an imperically decided correction. For instance: this sprint was 90 points and we had 22 points left. We should plan on having about 68 points total on our next sprint.

Once more, our team shouldn't feel bad about not finishing everything. Afterall, look at all the work we finished. But we should still have a discussion in our Retrospective to decide if we underestimated or over-planned.

[IMAGE 4] - The Timid Sprint

Congratulations! We completed all our planned work this sprint. But then we added more work and didn't complete everything. Let's talk during our Retrospective to see if we possibly *overestimated* some work, or if we just underplanned our sprint. Remember that even though it's great that we did all the things we said we would, we want to try to keep as close to the "Par Sprint" as possible for future data crunching and long term planning.

Daily Stand-ups

I'll keep this section short, since daily stand-ups are suppsed to be short as well. Meet up with your team and your BA/Scrum Master/etc. Go around and very briefly mention any tickets that moved from development to testing, etc. or any blockers you may have. Keep "technical solutions brainstorming" and "rubber ducking" out of this meeting and discuss on your own time. This is just to get an overview for the day and to keep people on track. No more than 10 minutes long.

Intro To JIRA

JIRA is a software product (one of many) that is built to track and visualize all of our tasks, epics, sprints, etc. With JIRA, we can build out teams, categorize tasks according to which epic they belong to, making our story point estimations, and it even generates a burndown chart for us.

We're just doing to jump right into it.

1. Visit https://www.atlassian.com/software/jira and click on the "Try it Free" button for Jira Software.
2. We're going to jump into the $10/month tier for now (it's a free trial - no worries!)
3. Fill out that form and let's go!
4. It's gonna ask us a few questions - Softward Development and Web Developer are the two answers we'll use.
5. Then we'll select that we're new to JIRA, new to Agile, we work on Features, and we have a flexible work schedule.
6. Select the Kanban template.
7. Name your project and continue on.
8. If it prompts you to take a little tour, do that.

Alright. Now we have a project made, but it's a little empty :(Not much of a project until we add some stuff to do and make it ours. Firstly, click on **Project Settings --> Features** and we're going to change some settings around. We need **Backlog, Sprints, Reports, and Estimation** all enabled. You can disable **Pages** if you wish. We won't use them right now. Then navigate back to your main project page.

It should look a little different now. Now we have a **backlog** section a sprint section.

- Backlog: where all your tasks are before they are added to a sprint during the sprint planning meeting. Feel free to add stories to your backlog whenever you feel. They'll be waiting you come planning time!

Real quick, navigate to "Board". We should see a few columns:

- To Do
- In Progress
- Done

Scroll all the way to the right side of that table and clickk the (+) plus sign button. Add a column called "In Testing" and drag it to the 3rd position (between In Progress and Done).

Okay, back in the backlog section, let's make our first task. I named mine "Learn to use JIRA a little better". Once you've created your first issue, click on it. In the description section is where our user story will go.

1. My user story will be "as a student I want to learn JIRA a little better so that I can practice Agile".
2. You can also put your Acceptance Criteria in the Descripttion section. There are other plugins to make it fancy, but this will work for now. Mine will be "I can make a detailed task and add it to a sprint without much effort".
3. A little below the description section is a spot to add your story point estimate. I estimated mine as a 3 for now.
4. Change the **Assignee** to yourself
    - Assignee: the person directly responsible for working on this task.
5. Add this story to the first sprint

Before we start our sprint, take a few minutes to make 1 more story. It can be anything you want it to be - just make sure to fill out all the fields and have a good user story, AC, estimation, etc. When you're done, add that story to the same sprint as the other one.

Once you've added another story, start the sprint. If there's a spot for it, add a **Sprint Goal** - which is just supposed to be 1 or 2 sentences, summarizing the work to be done; What do we aim to deliver in 2 weeks?

Cool. Now that your sprint is started, you have access to some of the reports, and you can go into the tasks and mark them as "In Progress", "In Testing", or "Done".

This is great and all - but it's pretty small scale. Let's add an Epic. Head over to your Roadmap and add an Epic. I named mine "User Authentication" and I changed the due date for 2 months from now. The cool thing about Epics is that you can add "Issues" to them. This is basically just telling JIRA which of our tasks work towards the completion of this epic. Click on the button to add an Issue and search for one of our tasks. Add it. Now when we go back to that task, it'll say that it's "**blocking**" that Epic we just made.

- Blockers: Tasks that need done before something else can be accomplished or worked on. Tasks can block other tasks (defining an order to work on them) or they can also block Epics, which just means that we can't say this Epic is completed until that task is done

In case you haven't noticed yet - a lot of these items can also have **labels** assigned to them.

- Label: A basic categorization (e.g. UX, Bug Fix, Admin)

Lab Activity

The Dice Game

- Get everyone a group of 5 people in a circle around a table. Everyone else can watch.

- There should be 20 dice on the table, all turned to #1
- Choose the order in which the "work" will flow. For example, Person #1 shouldn't be seated next to Person #2 (basically make it so the dice don't just go in a circle around the table - make it a little hectic).
- Explain that each person represents a different horizontal slice of the product:
    - idea/planning/brainstorming
    - development
    - code review
    - testing
    - push to production.
- Explain that we are a company and we have a HUGE product initiative that we are trying to develop and ship out. In order for our product to go from "Just and idea, all the way to production", each die must make it all the way to #6.
- Now it's time to start on our huge product initiative.
- Explain that the first person will turn all the dice so that the number #2 shows, then they slide them to the next person. Once they've done that, the next person will take all the dice and quickly turn them all to the #3. They then slide the dice to the next person who turns them all to #4 and so on until the last person has turned their die (turn 6's to 1's - in case you have more than 5 people playing)
- Point to the #3 person and tell them that they will represent a "bottleneck" in the process. In order for them to "complete their work" (aka pass the die to the next). The #4 must be showing, and the #1 side of the die must be facing away from them.
- Time to start! Keep a stopwatch going and time the group. How long did it take to get that huge product through all of the people? Mark that time down.
- Repeat the process - only this time, when the first person has turned 10 of the die, they should pass those 10 to the next person and then continue flipping the reamining 10 die. Once they've flipped the remaining 10 die, then they'll pass to the next person again.
- That next person can get started as soon as the first batch of 10 die are passed them. When they have filpped the 10 die, they should pass them to the next. In this way - we'll have multiple people flipping die at the same time.
- Stop the timer once the last person has flipped ALL 20 die. The time it took to get everyone to flip 20 die should be just a little bit quicker now.
- Run the whole thing again, only this time the first person can pass along batches of 5 die. Mark that time down.
- Run the whole thing one last time but this time - the batch size is 1 die. The first person should just flip 1 die, pass it along. Flip 1 die, pass it along.
- By the end of this 4th run - the time it takes for the entire batch of work to be done (everyone has done work on 20 die) should be minimal compared to the very first time. This is the key takeaway from this game.

- Explain that when we break our business problems down in to the smallest amount of deliverable work possible, things can move so much quicker.

"We broke that project down into more iterable sizes by reducing the number of die we needed to flip."

"We kept breaking these tasks down into smaller and smaller sizes until we reached a task which described a SINGLE PIECE OF FUNCTIONALITY - aka what Agile wants us to do with our software development."

This demonstrates that even when it feels like we're not getting much work done on an individual level in Agile, we're actually moving faster and getting the work done at a quicker pace. Through mastering the process, we can cut away at the 'in between' times and really get a quick turnaround time. We can really limit that "Application Delivery Lag" down to the length of a sprint. **From 3 years down to 2 weeks.**

Filling out a backlog

Let's hop back into JIRA and then create a new project (top left corner of the page is a dropdown menu where you can do this).

This project will be a Grocery List mobile application. I'll name off a couple Epics. It's up to you to order them in a sensible order, and then write a bunch of stories for each one. Estimate all your stories, and tag them with labels if applicable. If some stories NEED to be completed before others, make sure to add them as "Blockers", etc.

Household Sharing - I should be able to share my grocery list with members of my household and everyone should be able to use their accounts and edit the list

Basic List Functionality - a standard grocery list function

Personalized Template control - I should be able to make a template list that comes with items of my choice every time I make a new grocery list (aka items that I always need to get - eggs, wine)

Recipe Tracker - I should be able to add my own recipes to my account, so that when I edit my list, I can just click a recipe and it'll add all those items to my list.

Once you've made all of your stories - go back through them and try to split half of your stories in half. So if you had 50 stories total, take 25 of them and split those into 2 seperate stories. We want SMALL DELIVERABLES in Agile.

This will take quite a while. But the most important thing here is getting used to breaking problems and features down into smaller bits that we can ITERATE through in a cycle. This is how we make good, stable, software in a timely manner.

Week 1 Wrap Up

References:

1. https://dev.to/charanrajgolla/beginners-guide---object-oriented-programming
2. https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/
3. https://medium.com/@propagandra/understanding-how-code-is-executed-464c95e24b0b