Welcome to Software Engineering!

Are you interested in learning about software engineering?

Does finding out more about the software building process appeal to you?

Would you like to get more information about software development?

Does learning about the basics of programming interest you?

Would you like to know more about software architecture and design?

Have you ever wondered what the job prospects are like for a software engineer?

If so, this course is for you!

According to the most recent US Bureau of Labor Statistics report about 1,850,000

Play video starting at ::43 and follow transcript0:43

software engineer type jobs are posted annually in the US alone,

and job growth in the industry is projected to increase 22 percent through 2030,

At the time of this course publication, the median US software developer specialist salary

is 110,140 US dollars annually, and per hour pay is almost $53 US dollars.

Software engineering jobs are also a great way to

start on your path to a high-paying career in information technology.

This course has three instructors: Rav Ahuja, Bethany Hudnutt, and Lin Joyner.

Rav is a Global Program Director at IBM and the architect for several IBM Professional

Certificates. Bethany is an Instructional Designer at Skill-Up Technologies. And,

Lin is also an Instructional Designer at Skill-Up Technologies.

So, what’s in this course? It’s divided into five modules:

The Software Development Lifecycle

Introduction to Software Development

Basics of Programming

Software Architecture, Design, and Patterns

And Job Opportunities and Skillsets in Software Engineering

And when you complete this course, you’ll be able to

Describe software engineering and approaches to software development.

Identify software development tools and technologies,

Compare different types of programming languages and recognize basic programming constructs,

Outline approaches to application and deployment architectures,

And review the skills required in software

engineering and describe the career options it provides.

During this course, you will learn through:

guided instructional videos that walk you through key concepts, with essential need-to-know facts.

interactive exercises to reinforce what you’ve learned in videos,

insider viewpoints that enrich your understanding of IT support work.

practice assessments to help you gauge your knowledge, and,

finally, graded assessments to prove what you’ve learned.

Upon successful completion of this course,

you’ll earn a shareable badge and certificate that you can show to prospective employers.

We’re here to support your success.

If you have any questions for the course staff or

if you want to connect with your peers in this course,

Please feel free to post on the course discussion forums.

We’re excited you’re here. So, let’s get started!

Welcome to Meet the Experts.

In this course, you will learn from several experts in various roles, who will provide

their insights and perspectives about Software Engineering.

Let’s meet these experts and learn about where they work and what they do.

Hi, my name is Kadesha.

And I currently work as a developer advocate at GitHub.

My name is Liseidy Bueno.

I'm a Software Engineer and I work at Blend, also known as Blend Labs.

My name is Tommy Young.

I'm a software engineer at Google.

My role is Uber Technical Lead overseeing publisher video ads.

My name is Diane Panagiotopoulos.

I am a research and development engineer at Bandwidth.

And I work directly through Bandwidth, I am not self employed as like a contractor with

them.

Hi, my name is Zubin Pratap.

I'm a software engineer

and currently I'm a developer advocate at a Web3 startup

My name is Daniel Rudnitski, and I'm a full stack software engineer at IBM.

My name is James Reeve and I'm a full stack software engineer at IBM.

Hi my name is Rupinder and I work as a Chief Solutions Architect at Skillup Technologies.

Hello, everyone, my name is Upkar Lidder and I'm a developer advocate with IBM.

Hi, I'm Yann Stoneham.

I'm a solution architect at Smartronics.

Now, lets listen to some of these experts describe

a little more in detail about what they do.

Yes, so, a developer advocate is a software engineer

who enables developer to be successful with

a product.

That was a mouthful.

So, essentially, what that means is, for example, at GitHub, I create a lot of content.

So I will do blog posts, workshops, go to conferences to speak about GitHub, the product,

I will also showcase developers in the open source community, and to like,

give publicity to the projects that they're working on.

And being an advocate means that I'm able to communicate deeply about technical concepts

in a way that's digestible for engineers across experience level to understand.

Before I was an advocate, I worked as a software engineer in the finance sector

for two years.

I spent a majority of my time coding, I primarily worked on the front end

of the application, using tools such as Git, Docker, GraphQL, React, JavaScript, and the

very few times I would go into the back end,

we use Node.js on our back end to make changes and updates.

My experience as a software engineer, and also as a technical content creator, paved

the way for me to become a developer advocate at GitHub today.

I'm a software engineer right now.

I'm still pretty Junior.

So most of my tasks involve creating new features, fixing bugs, and addressing any customer concerns

that we have with that they have with the current software.

Yeah, so, I work on a suite of products that are

they're targeted to large content creators, video publishers, customers, such as ESPN,

Disney, etc.

We basically have suites of tools to help them monetize their video content, help them

distribute it, help them target it appropriately, report on it, troubleshoot it, for across

all devices, all surfaces.

So it's kind of the full suite of products, starts with the user interfaces that their

sales teams and trafficking teams use to sell the ad space,

upload the creatives, the advertisements, and, you know, target them to their audiences

and contextually to the content that they're, that they create.

We offer reporting on that.

So, a lot of its performance and various other analytics, reach analytics, and so forth.

Previously, before this, I was a software engineer at Google.

And before that, I was a software engineer at a small startup.

But before that, for about over 15 years, I

was a corporate lawyer, and then a business

executive for a while and even had my own startup in the middle.

So yeah, very diverse background.

And I taught myself to code in my very late 30s.

When my startup started to fail, and I really needed to keep it going, and my tech co-founder

quit.

So that's what I do.

That's who I am today.

I as a developer advocate, I not only write code, I also produce a lot of educational

content, and material that is useful for people to onboard onto Web3 and other technologies.

And my role is really to help connect the product team with the actual developer ecosystem.

Bring the two together through code, content, education, and community.

Welcome to “What Is Software Engineering.”

After watching this video, you will be able to define software engineering, list the responsibilities

of a software engineer, and compare and contrast software developers and software engineers.

Software engineering is the application of scientific principles to the design and creation

of software.

The field uses a systematic approach to collect and analyze business requirements in order

to design, build, and test software applications to satisfy those business requirements.

When computing began in the late 1950s, software engineering was a relatively undefined discipline,

but over time it transformed into a modernized engineering field.

The software engineering field became a discipline in the 1960s and evolved as new technologies

were developed and the approach to software development became more scientific.

Trends in software engineering transformed from ad hoc programming towards more formal

and standardized methods.

Initially, the creation of software lacked a formal development process.

As the world widely adopted computers, software became increasingly integral to more aspects

of life.

The inefficiencies in the software development process made it difficult to meet the rapidly

increasing demand for computing resources and complex software.

This led to what is known as the “Software Crisis” which began in the mid-1960s and

lasted until the mid-1980s.

During this period, software development often ran over budget, behind schedule, and consisted

of unmanageable, buggy code.

By the time older software solutions came to fruition, newer, better, and faster technologies

had already been developed, causing software engineers to have to refactor their code,

or completely redesign their system.

Often software development solutions that worked for small software systems did not

scale to large, complex projects.

Play video starting at :2:14 and follow transcript2:14

Now, Some of these issues still exist today, although to a much lesser extent due to the consistent

application of engineering principles to the software development process.

Computing resources have become more widely available and standardized methodologies for

software development allow for large, complex solutions that scale.

The solution to the “Software Crisis” involved transforming unorganized coding efforts

into an established engineering discipline.

The mid 1980s also saw a rise in the growth of computer-aided software engineering or

CASE which also helped to relieve the software crisis.

CASE tools can be divided into six categories: business analysis and modeling, development

tools such as debugging environments, verification and validation tools, configuration management,

metrics and measurement, and project management.

The term “software engineer” is often used interchangeably with software developer

but there are subtle differences.

Software engineers are also developers, but the term “software developer” is usually

deemed narrower in scope than that of a software engineer.

A software engineer’s knowledge is usually broader.

Software engineers take a systematic, big picture approach in their thinking to software

development whereas developers may have more creative approaches.

Both software engineers and software developers have specialized knowledge, but software engineers

use that knowledge to build entire systems whereas software developers use their knowledge

to write code to implement specific functionality within a system.

Software engineers are often employed on larger scale projects and they are focused on the

broad structure rather than solving an immediate problem.

Software engineers are tasked with designing, building, and maintaining software systems. Their

Responsibilities include writing and testing code, and consulting with stakeholders such as

clients, third party software vendors, security specialists, and other team members.

The adoption of a measured, scientific approach to software development has influenced the

way software is created and designed.

Today, the development process is typically guided by the Software Development Lifecycle

or SDLC.

The SDLC identifies the steps needed to develop high-quality software.

Later in this module, we will discuss the SDLC, the traits of high-quality software,

and roles common in the field of software engineering.

In this video you learned that:

Software engineering is the systematic approach to design and development of software.

Responsibilities of a software engineer include:

Designing, building, and maintaining software systems,

writing and testing code, and

consulting with stakeholders, third party vendors, security specialists, and other team

members. And finally,

software engineers build systems while software developers implement specific functionalities.

Welcome to Insiders’ Viewpoints: What is software engineering? In this video,

we will hear from experts discussing software engineering.

I think software engineering is the process of using programming languages and engineering

principles to build products. These products could look like web apps, mobile apps, desk top

apps, operating systems, or even networking systems. It is a skill that's integral to

the success of our modern society. Because without technology, we couldn't get a lot

of things done today. Within software engineering. It's a very broad term. And so within software

engineering, you have roles such as front end, back end, security, mobile, test, full

stack, DevOps, cloud data, and machine learning. And these are all different types of engineers

within software engineering.

I would define software engineering as building and improving software from anywhere from

design architecture to the user interface to fixing small bugs. So anything that falls

in that range, that's what software engineers do

So software engineering, to me, I think really has to do with, I consider it a creative process.

And it's fundamentally is the practice of designing, envisioning implementing, and then

supporting and maintaining software through the full lifecycle of it. So not just writing

code, but, you know, thinking about the software you're building before you even written a

line of code, the journey from idea inception to, to actual implementation and

launching upset software and then maintaining it, improving it, supporting it, and potentially

evolving into something else. So it's very much a sort of creative and full, full lifecycle

end to end engineering process.

So I know in the United States, at least software engineering, software development programming

they all have, depending on who you ask, they mean very specific things or very hazy things

or very contradictory things. Or it's like why do we even have three different titles

for the same thing. I think of those titles is kind of all meaning the same thing.

RECORD NARRATION:

What s the difference between a software engineer and a developer?

Yeah, I think the difference for me, I consider engineering to be a little bit sort of broader

software engineer be broader than then development, I think development to me means is sort of

one part of the process. So I think one fundamental difference for me is I think software engineering

is beyond just writing code, which I think a developer often is handed a task to, to

write a piece of code, create a a module, piece of software, whatever term you want

to use, I think engineering involves sort of starting, you know, a very long time before

you even get to the development process and continues on beyond the time you would actually

take your code, deploy it. So this kind of goes into the full lifecycle.

In my opinion, there isn't A difference really. So for example, I was trained as a full stack

engineer or a full stack developer, but I was hired as a software engineer to work on

the front end of the company's application. So I think it's a matter of semantics. And

a lot of it, I think, is a lot of gatekeeping. To keep people who don't know anything about

about the industry out. Somebody who has studied CS may argue differently from

I do, but as a person with a non traditional path into the world of software engineering,

I only see it as like title changes and title differences. And I think it depends on what

company you work in, and what company you work for. Because at the end of the day, we're

all using the same tools and technology to get the product shipped to production

. I think that they are pretty similar roles. They are used interchangeably a lot, I think.

But I think the main difference is that software engineers are also responsible for the system

design and architecture as well as data, how it's received, used, and how it's disseminated.

Whereas software developers, I think, focus more on the building of apps and features.

So, in Canada, being a computer programmer, or a software developer is incredibly different

than being a computer engineer or a software engineer, where to be an engineer, it means

you've taken specific classes in like codes of ethics. And you're held to the same ethical

standards as a civil engineer or a mechanical engineer. And so depending on where you are,

the name, the titles can mean very different things. And I know for purposes of immigration,

the titles also can mean very specific things. And you have to be careful about like, what

it is that you put down. But in job descriptions, I kind of see them all used interchangeably,

and I tend to just kind of think of them all as the same thing.

The, you know, it's much of a more of a longer time horizon, I think, in front in terms of

engineering versus development. I think there's a lot more considerations and hats you need

to put on throughout the software engineering lifecycle versus the kind of more narrowly

scoped development cycle.

Welcome to the introduction to the software development life cycle.

After watching this video, you will be able to

describe what the software development life cycle is,

explain its history, and

discuss some key advantages of using it.

The Software Development Life Cycle, known as the SDLC,

is a systematic process to develop high-quality software in a predictable timeframe and budget.

The goal of the SDLC is to produce software that meets a client’s business requirements.

The SDLC defines phases of the software development process that encompass their own process and

deliverables.

It is a cycle of planning, design, and development that can be implemented as an iterative approach

to software development.

Adherence to the SDLC minimizes risks and costs to the development of high-quality,

deployable software.

The software development life cycle began to take shape in the mid-1960s as software

development began to necessitate a more detailed approach because of its growing complexity.

The SDLC led to a more deliberate approach as large corporations needed to manage complex

business systems requiring heavy computational resources.

In its initial conception, it used what is called the “waterfall method” to manage

projects where the development of software follows a linear pattern through discrete

stages.

The SDLC has since been adapted, however, to more iterative methods in response to addressing

customer needs and shifting requirements.

Waterfall and other approaches to software development will be discussed in another lesson.

There are some key advantages for businesses in following the SDLC.

The first advantage is that it gives development teams a process to follow rather than using

an ad hoc approach to improve efficiency and reduce risks.

Secondly, there are discrete phases to the SDLC.

Each phase is well defined so that team members know what they should be working on and when.

The phases of the SDLC will be discussed in another video.

Because of the well-defined phases, it facilitates communication between the customer, other

stakeholders, and the development team.

The SDLC offers an overview of the process, so stakeholders know where they fit in to

that process.

Also, since each phase is discrete, cross-domain teams know when they have completed their

tasks and when development can move to the next phase.

The SDLC provides room for iteration where, at the end of a cycle, the process can circle

back to incorporate additional requirements as needed.

Problem solving is incorporated early in the cycle so problems are addressed in a timely

fashion and can be addressed in the design phase rather than during coding.

Finally, each team member has a well-defined role which reduces conflict and overlapping

responsibilities.

In this video you learned that

the SDLC provides a systematic process for software development.

Its initial development in the 60s and 70s was driven by the need for a systematic approach

because of the growing complexity of software.

Key advantages of the SDLC include:

A roadmap to the software development process, helping to reduce risk and improve efficiency,

Increased communication between the team and stakeholders,

Clearly defined and understood responsibilities for each team member, and

The ability to be used iteratively, allowing for changing requirements.

Welcome to the Phases of the Software Development Life Cycle.

After watching this video, you will be able to

name the phases involved in the SDLC,

describe each phase, and

identify several tasks associated with each phase.

There are generally six phases in the SDLC process,

planning, design, development, testing, deployment, and maintenance.

Each phase is discrete meaning that tasks from a previous phase do not overlap with

tasks in the next phase.

The original SDLC was conceived as a traditional waterfall method where the phases are linear,

but have since been adapted to introduce iteration so that shifting requirements can be accommodated.

Waterfall and other approaches to software development will be discussed in another video.

Note that some organizations may have different names for each stage.

For example, “planning” may be called “requirements” or “strategy” or “analysis”.

Also, some organizations may have additional or fewer stages.

In the first stage of the SDLC, the planning phase, requirements are gathered, analyzed,

documented and prioritized.

When planning a software solution, the following factors must be considered:

users of the solution

the overall purpose of the solution,

data inputs and outputs,

legal and regulatory compliance,

risk identification,

quality assurance requirements,

allocation of human and financial resources, and

project scheduling.

As part of the planning process,

labor and material costs are estimated and

weighed against time constraints.

Also, project teams are identified, and roles of each team member are proposed.

If stakeholders are struggling to define requirements, often the development team may produce prototypes

during the planning stage to tease out those requirements.

A prototype is a small-scale replica of the end product used to get stakeholder feedback

and establish requirements.

A prototype is used to test basic design ideas.

Though prototyping usually occurs during the planning stage, prototyping can occur at various

phases of the SDLC whenever requirements need to be reconsidered or clarified as the project

develops.

After requirements have been gathered, they are combined into a document called a

software requirements specification, or SRS, document.

The SRS needs to be clearly understood and approved by all stakeholders.

The developers are also involved at this stage so they can gain a clear understanding of

these requirements.

Requirements and the SRS will be discussed in more detail in a later video.

In the design phase, the requirements

gathered from the SRS are used to develop the software architecture.

Several team members work together at this stage to design the architecture.

The architecture is reviewed by the stakeholders and team.

And during this phase, prototypes can be designed.

A prototype is a preliminary mock-up of the system, or parts of the system, used for demonstration

purposes.

The document created in this phase is called a design document, and is used by developers

during the next phase, which is the development phase.

The development phase, sometimes called the “building" phase or the "implementation"

phase, is when the developers start the coding process once the design document is completed.

The project planners use the design document to determine and assign coding tasks.

This phase often requires the use of programming tools, different programming languages, and

software stacks.

Organizations may also have standards or guidelines that need to be followed.

The testing phase is next in the process once the coding is complete.

And some large projects have dedicated testing teams.

Code needs to be thoroughly tested to ensure it is stable, secure, and meets the requirements

outlined in the SRS.

Testing can be manual, automated, or a hybrid of both.

Product bugs are reported, tracked, and fixed, and code is retested until the software is

stable.

Some common levels of testing include

unit testing,

integration testing,

system testing, and

acceptance testing.

Each of these testing levels will be described in further detail in another video.

The deployment phase is where the application is released into the production environment

and made available to users.

This can also happen in stages—

first, it is released onto a user acceptance testing, also called UAT, platform and once

the customer signs off on the functionality,

it is released to production.

This approach can be used for making software available on a website, mobile device app

store, or a software distribution server on a corporate network.

Finally, the maintenance phase happens once the code has been deployed into a production environment.

This phase helps to

find any other bugs,

identify user interface issues, or UI for short, and

identify other requirements that may not have been listed in the SRS.

Code enhancements can also be identified at this stage.

If bugs are discovered in this phase that were missed during testing, these errors may

need to be fixed for high-priority issues or incorporated into the requirements as part

of a future software release and the process can start over again.

In this video, you learned that

The SDLC can be divided into six phases. Planning involves requirement gathering and development

of the SRS.

The architecture is developed during the design phase and the design document is created.

The Development phase is when coding takes place, and then

during the testing phase issues with the code are found and fixed if possible.

Deployment is when the code is released to the production environment.

And finally, in the maintenance stage feedback is collected from stakeholders, other UI issues

may be identified, and code enhancements suggested.

And this information then can be fed into another software development cycle if necessary.

Welcome to Building Quality Software.

After watching this video, you will be able to:

list common software engineering processes and

describe the common software engineering processes required for building-high quality software.

There are numerous processes that are common to software engineering projects. In this

video we will discuss six of them:

requirements gathering,

design,

coding for quality,

testing,

releases, and

documenting.

The software requirements specification, or SRS, encompasses the process of collecting

and documenting the set of requirements that the software needs to adhere to.

It may include a set of use cases that describe the business needs and user flows that the

software must implement.

Software requirements can be classified into four broad categories:

functional,

external and User Interface, or UI,

system features,

and non-functional.

These categories are discussed in more detail in another video.

Software design is the process of transforming the requirements into a structure that is

implementable using code.

The software design process translates the requirements into a language the developers

can use to write the code. It transforms the requirements into a software solution.

The technical lead breaks down requirements into sets of related components with clearly

defined behaviors, boundaries, and interactions. These components define the system architecture.

The system design incorporates guidance on system functions, performance, security, and

platform characteristics.

The design communicates business rules and application logic, application programming

interface design, which is how apps talk to each other or communicate with the database,

user interfaces, and database design.

Code quality refers to the characteristics of the code including attributes such as

maintainability,

readability,

testability,

and security.

Quality code must fulfill the intended requirements of the software without defects.

Additionally, it should be

clean and consistent,

easy to read and maintain,

well documented,

and efficient.

Coding for quality entails following a set of coding practices during development. These

include:

following common coding standards, conventions, patterns and styles,

using automated tools, known as linters, to detect programmatic and stylistic errors, and

commenting in the code itself to make it easy for others to understand and modify.

Software testing is the process of verifying that the software matches established requirements

and is free of bugs.

Its purpose is to identify errors, gaps, or missing requirements when compared with stated

requirements.

Properly tested software ensures reliability, security, performance, and efficiency.

Software testing can often be automated or done manually.

Levels of testing include unit, integration, system, and user acceptance.

Unit testing is often done by the developer and tests the smallest component of code that

can be isolated from the rest of the system.

Once the components are integrated into the larger product, integration testing occurs.

Then, after the larger product is deemed completed, system testing can take place.

User acceptance testing, or UAT for short and sometimes called beta testing, is when

the software is tested by the intended end user. Types of testing can broadly be divided

into three categories, functional, non-functional, and regression. Testing levels and types will

be further explained in an upcoming video.

When the newest version of the software is distributed, it is referred to as a “release.”

Different types of releases are intended for different audiences.

There is generally an “alpha,” a “beta,” and a “GA” release. GA stands for general

availability.

The alpha release is the first functioning version of the system released to a select

group of stakeholders.

The alpha release likely contains errors and

may not contain the full feature set but does contain most of the desired functionality.

Design changes may still occur during this release stage.

The beta release, also called a limited release, is given to the stakeholders outside of the

developing organization.

One of the intents of the beta release is to try out the software under real conditions,

test the functionality, and identify any outstanding bugs or errors.

The beta release should meet all the functional requirements.

Then, after beta release changes are agreed upon, made, and tested, and a stable version is released.

The audience for the GA release is all users.

Software documentation should be provided to both non-technical end-users and technical

users.

Play video starting at :5:26 and follow transcript5:26

System documentation is geared towards the technical user. Technical users may be other

engineers, developers, or architects. System documentation explains how the software operates

or how to use it. It consists of README files, inline comments, architecture and design documents,

verification information, and maintenance guides.

Play video starting at :5:50 and follow transcript5:50

User documentation is provided to the non-technical end-users to assist them in the use of the

product. Generally, user documentation is provided in the form of user guides, instructional

videos and manuals, online help, and inline help.

Play video starting at :6:6 and follow transcript6:06

More details about documentation will be discussed in another video.

In this video, you learned that:

Requirement gathering is collecting and documenting the set of requirements that the software

needs to adhere to.

Designing transforms requirements into a structure that developers can use.

Coding for quality entails following a set of coding practices during development.

Testing is the process of verifying that the software matches established requirements

and is free of bugs

There are three types of releases including: alpha, beta, and general availability. And finally,

documenting requires text or video that explains the software to technical and non-technical

users.

Welcome to Requirements.

After watching this video, you will be able to:

Describe the steps of the requirement gathering process.

Explain the purpose of a User Requirement Specification, or URS, document.

Explain the purpose of a Software Requirement Specification, or SRS, document, and

Explain the purpose of a System Requirement Specification, or SysRS, document

Play video starting at ::32 and follow transcript0:32

Requirement gathering is a six-step process of defining a problem to

be solved and documenting how to go about solving that problem. These steps include:

identifying stakeholders,

establishing goals and objectives,

eliciting requirements from the stakeholders,

documenting the requirements,

analyzing and confirming the requirements, and

prioritizing.

Generally, the stakeholders work for the organization that

requests the development of the software product.

Key personnel from the organization may include

decision-makers,

end-users,

system administrators,

engineering,

marketing,

sales,

and customer support personnel.

It is good to have a representative from every group that the product affects.

The goals of the product should be clearly defined.

Goals are broad, long-term achievable outcomes.

Goals can include customer outcomes and business goals.

Next, objectives should be identified. Objectives are more specific than goals

and they are actionable and measurable actions that achieve the stated goals.

The next three steps, eliciting, documenting,

and requirement confirmation are usually completed iteratively.

Elicitation can be accomplished through

surveys,

questionnaires,

and interviews. As the requirements emerge, they should be

documented and checked to ensure they

align with the goals and objectives.

Documented requirements should be easily understood by stakeholders and the project team.

In order to confirm the requirements,

they should be analyzed to ensure consistency,

clarity,

and completeness. And after analysis, the requirements should be shared with

and approved by the stakeholders.

After confirmation, requirements should be prioritized. Labels such as

“must-have,”

“highly desired,”

and “nice to have” are helpful. If possible, order the requirements within those categories.

Play video starting at :2:29 and follow transcript2:29

Typically, there may be three documents that result from the requirements gathering process:

software requirements specification, or SRS,

user requirements specification, or URS,

and system requirements specification,

or SysRS. The most common of these is the software requirements specification.

The software requirements specification, or SRS,

is a document that captures the functionalities that the software should perform

and also establishes benchmarks or service levels for its performance.

Parts of an SRS include:

A purpose statement that contains

the intended use of the SRS,

its audience and

scope,

constraints, assumptions and dependencies, and

requirements, which can be sorted into four categories:

Functional requirements

External Interface requirements

System Features

and Non-functional requirements

The product’s purpose describes

who will have access to the SRS and

how they should use it.

The scope describes

the benefits of the software,

its goals, and

objectives.

The second part of the SRS should detail constraints, assumptions, and dependencies.

Constraints describe how the product must operate under given conditions that may

limit options in the design phase such as confirmation to standards or hardware limitations.

Assumptions may include things like a required

operating system or hardware that is needed by the software to function.

Dependencies on other software products should also be noted.

Requirements can be classified into four categories.

Functional requirements are those that cover the functionalities of the software.

External requirements are the requirements that address the behavior of the software

in relation to external entities such as users and interactions with other hardware or software.

System features are a subset of functional requirements.

These are required features for the system to function.

There are also non-functional requirements such as specifying performance,

safety, security, and quality standards.

User requirements describe the business need and expectations

of the end-users from the software system.

The user requirements are written as “user stories” or

“use cases” that answer three questions:

Who is the user?

What is the function that needs to be performed?

And why does the user want this functionality?

User acceptance testing determines if these requirements have been met.

Often though, the user requirements and software requirements are combined

into a single SRS document. The SRS details the expectations of the software system.

The System Requirement Specification document, or SysRS, to differentiate it from

the SRS, clearly outlines the requirements of an entire system. The system requirement

specification is often used interchangeably with software requirement specification,

but the SysRS is actually broader in scope than the SRS.

Many software projects develop an SRS rather than a SysRS.

The SysRS contains

system capabilities,

interfaces, and user characteristics. It also may include

policy requirements,

regulation requirements,

personnel requirements,

performance requirements,

security requirements,

and system acceptance criteria.

It also outlines expectations of the hardware needed for the system

in addition to software requirements.

Play video starting at :6:9 and follow transcript6:09

In this video you learned that:

The requirement gathering process entails identifying stakeholders, establishing

goals and objectives, and eliciting, documenting, confirming, and then prioritizing requirements.

The SRS documents functional, external, system, and non-functional requirements.

The URS documents user stories.

Play video starting at :6:33 and follow transcript6:33

And finally, the SysRS documents system capabilities and acceptance criteria,

and policy, regulation, personnel, performance, security, and hardware requirements.

Welcome to Software Development Methodologies.

After watching this video, you will be able to

list several commonly used approaches to the software development life cycle,

explain waterfall, V-shape model, and Agile methods, and

compare the pros and cons of each of these three methods.

There are many ways to approach software development.

A specific methodology for developing software is commonly used in order to assist the development

team to clarify communication among team members and determine how and when the information is

shared.

In this video, we will discuss three of these approaches:

Waterfall,

V-shape model,

and Agile.

In the beginning, when the SDLC was conceived, it implemented what is known as the

waterfall method.

Waterfall is a sequential method of software development where the output of one phase

is the input for the next phase of the cycle.

Development and work on the next phase start only after the completion of the previous

phase.

All planning, such as defining requirements and architectural design, is done up front.

The customer usually does not see the product until it is in the testing phase.

For a major version release of the product, the same process is repeated resulting in

long intervals, such as years, between releases.

The V-shape model is named as such because the phases form the shape of a V.

The phases going down the left side of the V are called “verification".

Then, going up the right side of the V, those phases are called, "validation."

The V-shape model is like waterfall in that it is also sequential.

Each phase in verification corresponds with a validation phase.

There are four stages that occur on each side of the V. Going down the V are planning, system

design, architecture design, and then module design.

The bottom of the V is the coding phase.

And going back up the V are the four phases that correspond to the phases going down the V:

unit testing,

integration testing,

system testing, and

acceptance testing.

The tests are written during the verification phases on the left and executed during the

validation stages on the right.

Now, the Agile model is different.

It focuses on a collaborative software development process over multiple short cycles rather

than a strictly top-down linear process.

Agile is what is called an iterative approach to development.

It still aligns with the SDLC, but each phase is short.

Teams work in cycles, or sprints, which are usually one to four weeks long.

Unit testing happens in each sprint to minimize the risk of failure.

Rather than the “maintenance” stage of the SDLC, the final stage of the sprint is

a feedback stage.

At the end of each sprint, a chunk of working code is released at a meeting called the “sprint

demo” where stakeholders can see the new functionality and provide feedback.

After the sprint demo, the entire process is repeated for every sprint cycle.

After several sprint cycles, a minimum viable product, or MVP, is developed so stakeholders

can provide feedback on the basic feature set.

The MVP contains a feature set to validate assumptions about the software.

The four core values of Agile development outlined in what is known as the "Agile manifesto"

are:

individuals and interactions over processes and tools

working software over comprehensive documentation

customer collaboration over contract negotiation, and

responding to change over following a plan.

The main difference between traditional SDLC methods such as

waterfall and the v-shape model compared to the

Agile method of software development is the former are sequential whereas

Agile is cyclical.

Traditional SDLC methods, such as waterfall and V-shape, center around the whole product

being developed before soliciting customer feedback, whereas

Agile focuses on quick, short bursts of development.

There are pros and

cons to each method, though Agile is probably the most popular method used in modern software

development.

Regarding the pros of the waterfall method, it is easy to understand and follow.

Each stage is discrete and well-defined, making it easy for all team members to understand

their roles.

Also, since planning is done upfront, it is easier than iterative methods to estimate

a budget and allocate resources.

That said, waterfall lacks flexibility.

Since all planning is done upfront if a requirement is changed or overlooked that change can be

hard to incorporate at a later date.

Inevitably, unforeseen complications happen, or agreed upon functionality shifts from what

was initially envisioned.

Like waterfall, the V-shape model is simple and easy to use.

It is even more rigid than waterfall but

designing test plans during the verification phase saves considerable time during coding

and validation phases.

Drawbacks are also similar to waterfall because it

does not readily accommodate changing requirements.

Once an application is in the testing phase it is extremely difficult to go back and change

functionality.

Agile development is different, relying on ongoing research, planning, and testing during

product development.

When adding new features to a project, development still goes through the same phases as in traditional

SDLC,

but with Agile, new, and changing requirements are handled quickly and easily because planning

is initiated at the beginning of each sprint cycle.

Most resources are spent on the building phase.

At the end of each cycle, the QA team, stakeholders, and the customer have some piece of working

code to test against requirements and are encouraged to provide feedback.

As coding languages and technologies have developed in recent years, they now allow

for modular design, where developers can focus on smaller chunks of code that are readily

integrated into the larger product.

These small chunks can be released to provide the MVP.

Cons of Agile are that upfront planning such as budgeting and scheduling can be challenging

because the

overall scope of the product is not clearly defined.

In this video, you learned that:

Three of the common approaches to software development include waterfall, V-shape model,

and Agile

waterfall and V-shape are sequential whereas Agile is iterative

both waterfall and V-shape models are easy to implement but neither accommodates changing

requirements well and

Agile allows for changing requirements but resource allocation can be challenging.

Welcome to Software Versions.

After watching this video, you will be able to:

Discuss software versions on computing platforms

and identify software versions and numbering.

Software versions tell us a lot about programs and applications.

Users can determine what software version they are using, and developers can provide

useful information with version numbers.

Software version numbers vary in length and meaning; however, most version numbers follow

a similar format and represent similar information.

Version numbers indicate when the software was released, when it was updated, and if

any minor changes or patches were made to the software.

Software versioning is how software developers keep track of new software, updates, and patches

for programs and applications.

Version numbers can be displayed in several ways.

Version numbers can be short or long, depending on the software and the preference of the

developer, with 2, 3, or 4 number sets.

Each number set is divided by a period.

The first release of an application or program might have a 1.0 as the version number to

indicate no updates, patches, or fixes to the software.

Note: A version still in beta or testing could have a version number lower than 1, such as

0.9.

A program or application with many releases and updates will have a longer number, sometimes

4 different number sets within the version number

Some software developers may use dates for their versioning.

For example: Ubuntu Linux version 18.04.2 was released in 2018 April.

The third number set, point-2, designates an additional change or update.

What do these numbers mean?

Some version numbers follow the semantic numbering system and have 4 parts separated by a period,

but not all numbering systems follow this 4-part example.

In semantic numbering, the first number indicates major changes to the software, such as a new

release.

The second number indicates that minor changes were made to the software.

The third number in the version number indicates patches or minor bug fixes.

Finally, the fourth number indicates a build number or a build date, and it can indicate

less significant changes made.

Software version numbers are identified in the About or Help section of software.

You can practice identifying your version number in a web browser.

This example illustrates how to view the version number in the Google Chrome desktop web browser;

however, if you are using a different web browser, you should be able to follow the

same or similar steps.

To view the browser version:

First, select the three dots or three lines in the top-right corner of your browser.

Next, select the menu item “Help.”

Then, select “About” to view the version information.

The version of your web browser will display.

Are newer versions of software compatible with older versions?

Lack of compatibility between old and new versions of software is a common problem.

You can troubleshoot compatibility issues by viewing the software version to determine

if you are using an outdated version of the software.

Sometimes updating software to a newer version will resolve compatibility issues.

Some software is backwards compatible.

If a program or application is backwards compatible, then the older versions of files, programs,

and systems will work properly with newer versions.

Play video starting at :3:52 and follow transcript3:52

In this video, you learned that:

Version numbers indicate the history of changes, updates, and patches to software,

Some version numbers follow the semantic numbering system and have 4 parts separated by a period,

Compatibility with old and new versions of software is a common problem,

and you should view the version of the software you are using to determine software compatibility.

Welcome to Software Testing.

After watching this video, you will be able to:

Define the terms functional testing, non-functional testing, and regression testing, and

compare and contrast typical testing levels.

Software Testing is the practice of integrating quality checks throughout the software development

cycle.

The purpose of testing is to

check whether the software matches expected requirements and

ensure error-free software.

In order to test software, the team writes “test cases.”

These test cases are written to verify the functionality of a software application

and ensure requirements have been satisfied.

Test cases can be written in different stages of the SDLC and may vary depending on the

type of test or the method used to develop the software, such as Agile or waterfall.

A test case contains:

steps,

inputs,

data, and

the expected corresponding outputs. Regardless of the test type or development method, test cases

should always be written after requirements are finalized.

Software testing helps evaluate the software to identify whether or not the software product

meets requirements and is error-free.

Types of testing can be broadly classified into three categories:

Functional testing,

Non-Functional testing,

and Regression testing

Functional testing usually involves black box testing which is a method of testing

without looking at source code or internal structure.

Functional testing is only concerned with inputs and corresponding

outputs of the system under test, also called the SUT. It is entirely based on testing functional

requirements.

Functional testing can be carried out manually or

using automated tools.

The goal is to test the functionality of the application making sure the application is

usable and

accessible. Functional testing tests the SUT, to make sure it meets functional requirements.

Functional testing makes sure that when user errors or input edge cases do occur,

the software handles those exceptions seamlessly by

displaying appropriate error messages.

Non-functional testing includes testing the application for attributes like

performance,

security,

scalability, and

availability. Non-functional testing checks to see if the SUTs non-functional behavior

is performing properly.

Non-functional testing should answer questions like the following:

How does the application behave under stress?

What happens when many users log in at the same time?

Are the instructions in documents and user manuals consistent with the application’s

behavior?

Does the application behave similarly under different operating systems?

How does the application handle disaster recovery?

And how secure is the application?

Regression testing, also called maintenance testing, confirms that a recent change to

the application,

such as a bug fix, does not adversely

affect already existing functionality.

Regression testing should occur when there has been a change in requirements or when

defects have been fixed.

In order to conduct regression testing, all or some of the test cases should be selected

to test against the application. Regression test case selection and prioritization can

be challenging and can depend on several factors.

Common reasons for regression test case selection include cases that:

have frequent defects ,

contain frequently used functionality,

contain features with recent changes,

or are complex test cases,

edge cases, and

randomly successful or failed test cases.

Now that we have discussed different types of testing, let’s discuss testing levels.

There are four testing levels:

unit,

integration,

system, and

acceptance.

Each level occurs at a different time in the SDLC. There are 4 different levels in order

to reduce the amount of time spent on testing by preventing overlap. We will discuss each

of these testing levels next.

Unit testing refers to tests that verify the functionality of a specific section of code,

usually at the function level.

It is performed by the software developer or engineer during the development phase of

the software development life cycle.

Unit testing aims to eliminate construction errors before code is integrated with other

modules. Unit testing is intended to increase the quality of the resulting software as well

as the efficiency of the overall development process.

Integration testing

seeks to identify errors when two or more smaller, independent code modules are combined.

Integration testing is another type of black-box testing.

Prior to integration testing, smaller, independent code modules that passed unit

testing are incorporated into the larger software application.

After modules are integrated together, then integration testing can occur.

Integration testing exposes bugs that occur when those smaller units of code interact

with each other.

Integration testing uncovers deficiencies in communication with

a new module in conjunction with

other existing modules,

databases, or

external hardware. Integration testing uncovers situations where bugs develop due to differing

programming logic between modules, for instance. Also, sometimes during module development,

requirements change, and the module isn’t fully unit tested. Poor exception handling

can cause problems when modules are integrated together.

System testing occurs after integration testing and is conducted on a complete, integrated

system to

evaluate the system's compliance with its specified requirements.

It validates the system as a fully completed software product.

System testing is both functional and non-functional.

System testing is done in a staging environment, which should be similar to the production

environment. And finally,

acceptance testing is formal testing with respect to user needs, requirements, and business

processes. It determines whether a system satisfies the needs of the

users,

customers, and

other stakeholders. Acceptance testing is usually done by the customer or the stakeholders

during the maintenance stage of the SDLC.

In this video you learned that:

There are three categories of testing: functional, non-functional, and regression .

Unit testing verifies small, independent chunks of code.

Integration testing looks for errors when two or more small chunks of code are combined.

System testing validates the system as a fully completed software product and

acceptance testing verifies correct implementation of user requirements and business processes.

Hello and welcome to Software Documentation.

After watching this video, you will be able to:

List documentation formats.

Compare and contrast product documentation to process documentation.

Describe the categories and types of documentation.

And explain the purpose of standard operating procedures.

Software documentation is information about the software that describes what the product

is and how to use it. These can be

written,

video, or

graphical assets associated with a software product’s development and use.

Documentation can be in any of these three formats.

Documentation is an essential aspect of software engineering applicable across all the phases

of SDLC.

Software Documentation can be written for different types of audiences – such as end

users, software developers, QA engineers, system administrators and other stakeholders.

Documentation can be divided into two categories, product and process.

Product documentation relates to the product’s functionality, whereas process documentation

describes how to complete a task.

Process documentation should provide the requirements for the quality implementation of a business

process.

Now, let’s discuss some specific types of product documentation.

There are many types of documentation, and we will discuss five categories including

requirements,

design,

technical,

quality assurance,

and user documentation.

Requirements documentation is written during the planning phase of the SDLC and is intended

for the development team including the developers, architects, and QA personnel.

Requirements documentation describes the expected features and functionality of the software

system.

It includes the software requirements specifications, system requirement specifications, and user

acceptance specifications.

Design documentation is written by the software architects and the development team to explain

how the software will be built to meet the requirements.

It consists of both conceptual and technical design documents.

Technical documentation includes comments written in the code to help other developers

read the code to understand its behavior. It also may include working papers that explain

how the code works and documents that record engineers’ ideas and thoughts during project

implementation.

Quality assurance documentation includes all documents that pertain to a testing team’s

strategy, progress, and metrics.

Types of test documentation include test plans, test data, test scenarios, test cases, test

strategies, and traceability matrices. Traceability matrices map test cases to their requirements.

User documentation is intended for end-users and explains how to operate the software or

help them to install or troubleshoot the system.

End-user documentation includes frequently asked questions, installation and help guides,

tutorials, and user manuals.

Standard operating procedures, called SOPs, often accompany process documentation.

Process documentation provides an overview of a process, but SOPs go through much greater

detail. The SOP is written documentation that explains step-by-step how to accomplish a

common, yet complex task that is organization specific. For example, checking in code using

a code repository is common knowledge for a software engineer.

However, an organization might have specific steps to follow for that organization in order

to get code merged into the main branch.

The SOP documentation explains those steps in detail. SOPs can be in the form of a

flowchart,

a hierarchical outline,

or step-by-step instructions.

Documentation, in any form, must be kept up to date. Take for instance online user manuals.

If a cloud-based application user interface changes, then the accompanying online documentation

must be updated accordingly. Businesses need to ensure they allot resources for this step.

With regards to the software development and the SDLC, updating documentation happens during

the maintenance phase.

Ideally, documentation should also be reviewed periodically to ensure its accuracy.

In this video you learned that:

Documentation comes in three formats: written, video, or graphical.

Process documentation describes how to complete a task.

Product documentation relates to how a product functions.

The types of product documentation include requirements, design, technical, QA, and user.

And SOPs are written instructions detailing an organization's specific procedure.

Welcome to Roles in Software Engineering Projects.

After watching this video, you will be able to

list the common roles on a software engineering project,

describe each role, and explain the responsibilities of each role.

There are several common roles on a software development project. And these roles can have

different names depending on the approach being used, such as Agile or waterfall. Sometimes

different companies have different names for similar jobs. But, not all projects will have

all these roles. The roles we will discuss in this video are

project manager or scrum master, stakeholder,

system or software architect, UX designer,

software developer, tester or QA engineer,

site reliability or Ops engineer, product manager or owner, and

technical writer or information developer. Now let’s describe each role and some of

the responsibilities for each job.

Traditional SDLC methods have project managers but in Agile the equivalent

role is called a Scrum master. A project manager makes sure

the project runs smoothly and facilitates communication about the project. The project

manager often deals with bigger picture issues such as:

Planning, scheduling, and budgeting; Allocating personnel and resources;

Executing the software plan; and Team communication.

In Agile, there is a Scrum master. Rather than focusing on planning, the Scrum master

is focused on ensuring team and individual success. Remember that the four

core Agile values prioritize people and communication over

process, and the Scrum master is responsible for facilitating that communication.

The stakeholders are the people for whom the product is being designed.

They include individuals such as the customer, end-users, decision-makers, system administrators,

and other key personnel. The stakeholder is mainly responsible for

defining project requirements and providing feedback if the team members need clarification

on requirements or if a proposed solution cannot be solved as planned.

The stakeholders may also sometimes participate in beta testing and acceptance testing before

the software is released.

The system architect, designs and describes the architecture of

a project as well as communicates that architecture to team members.

They are sometimes also called a software architect or a solution architect.

They are responsible for designing the essential characteristics of

the inner structure and technical aspects of the software. The architect

provides technical support across the different stages

of the SDLC.

Note: software architecture will be discussed in further detail in another module.

UX means user experience. The goal of a UX designer is to balance making

the software intuitive but also as robust as it needs to be to address requirements.

They define how the software behaves from the user’s perspective.

The UX designer determines how the software communicates its functionality

to the end-user and how the end-user interacts with it.

Next, the developers write the code that powers the software.

Responsibilities include implementing the architecture laid out in

the design document, incorporating the requirements laid out in

the software requirements specification, and employing the UX requirements determined by

the UX designers.

Testers or QA engineers are in-charge of ensuring the quality of the product

and that the software solution meets customer requirements.

They are responsible for writing and executing test cases to identify

bugs or deficiencies and provide this feedback to the development teams.

A site reliability engineer, sometimes called an SRE or ops engineer,

bridges development and operations by combining software engineering expertise with IT systems

management. They track incidents and facilitate meetings

to discuss them. They also automate systems, procedures, and

processes; assist with trouble shooting; and

ensure reliability for the customer.

The product manager or product owner has the vision of what the product should look like.

They have an intimate understanding of the client’s requirements,

and the end-user’s needs. They are responsible for

leading development efforts to create the software and

for ensuring the product provides the value stakeholders are looking for.

Finally, the technical writer or information developer

writes documentation for the end-user. They write documentation on

technical material geared towards a non-technical audience. Not only does this documentation

help the end-user to use the software, but it also helps the customer so they can provide

timely feedback to the development teams. Technical writers may be asked to write

user manuals, reports,

white papers, and press releases.

In this video you learned that: There are a variety of job roles on a software

development project and each occupation has responsibilities unique to each role.

Welcome to Insiders’ Viewpoints: Job Roles in Software Engineering Teams.

In this video we will hear from experts discussing the various job roles you can expect to work

with on your Software Engineering team.

On the engineering team I worked with, we had a product manager, we had a tech lead,

we had a QA, we had a few QA analysts or test engineers.

And we also had access to UX designers.

And so the product manager ensured that we were on task and on track to deliver the features

that we said we were going to build according to the business needs, that way we're not

left behind.

Play video starting at ::45 and follow transcript0:45

Yeah, I think, in my experience, the roles that the software engineers would interact

with the most, I think would, you know, in no particular order product managers, so the

folks who are actually sort of working with the sales teams, the marketing teams, figuring

out what actually needs to be built, what our customer needs, what are innovations and

ideas that, you know, looking forward, that we might, you might want to invest in and

build out.

We asked our experts to describe these job roles in software engineering teams in further

detail

I think software engineering these days, as it becomes more and more, more more and more

large scale, more complex more systems at play.

More considerations, there's just a lot of things to consider in today's in today's world

as a software engineer, so you often work with a project or program manager who helps

you set up your timelines, helps you ensure you're meeting your goals, helps

you unblock yourself helps you keep your stakeholders community up to date with what is what is

happening with your project.

So there's just a lot of pieces involved.

And I think project and program managers really are there to help tie it all together.

So the product manager would always act somewhat like a scrum master.

So like, we would have stand ups.

And we would talk about the tickets that we had in JIRA.

And it was it was like, okay, so what are you working on?

What do you need help with?

And are there any issues.

And during that process, the product manager does ensure that we were on task and on track,

and if we had any issues, he would help us to resolve it.

Play video starting at :2:26 and follow transcript2:26

You'll have a software engineer who's more focused on the architecture and they can keep

the the big picture in their brain more easily.

And then some software engineers who are more than nitty gritty, I'm sitting down and I

am writing code and my my job is to like, turn business logic into computer logic.

And some companies have those all split out as different roles.

I think software engineers, and then these roles that the more they're embedded and work

closely together and actually collaborate as part of a single team.

versus handing off things in a more waterfall approach.

I think that that's generally the the best way, or was one of the most effective ways

is to actually work together as much as possible regular check ins, actually having, you know,

brainstorming sessions, group chats that involve all the various folks, all the roles, I think,

close collaboration, there is super key

And at some companies, all the different hats have their own titles.

So you might be talking to a software engineer who's actually doing more kind of DevOps infrastructure

building, monitoring and alerting kinds of work.

And then there's a software engineer who's also doing QA work.

And they're the ones who are focusing on pipelines and continuous continuous development, continuous

integration type projects, working on getting test suites and automation all set up.

In terms of working with the UX designer or the UX engineer, they would give me like specific

specs from figma.

In order to meet the certain specifications of the business, it's let's say, we're building

a checkout page.

So the UX UX designer would design the spec page and figma and then deliver that spec

to me, and then I would code it according to the design.

UX designers who actually come up with it, if you work, if you're working on an engineering

or user interface, you work with UX designers to, to essentially from the moment of sketching

out sort of a, you know, a draft on paper or whiteboard of an idea all the way down

to working with them on mocks and implementing the mocks iterating.

On those UX researchers often are worked with when you actually need to go out and do research

and market and interview your partners or customers to get sort of as real time as possible.

And just, you know, real feedback from real users of your, of your software.

So once I code something, I get it reviewed by my tech lead or my senior engineer, it

would go to QA for them to test the application to ensure that whatever I did or implemented,

didn't break anything.

If something is broken, that I didn't catch, the QA tester would write up like a test plan

or would do like a detailed testing document.

That way I understood exactly what needed to be resolved.

And I would resolve it, and then go back to the QA person and say, Hey, this is fixed,

can you take a look at it again, and they would tell me whether or not it's fixed and

whether or not something else was broken, which often happens with code.

So you know, you generally work a lot with product managers, because you need to know

what to build.

And they generally give you a lot of guidance on, on what they want to build, and you collaborate

with them in terms of how to build it and discuss trade offs and timelines, and expectations.

So it's a very much a collaborative process.

Typically, I have found that I interact most with product owners.

So either at some places, it's been a customer who works inside the company.

And so I go to them quite frequently for questions on clarifying refinements for getting test

schedules up and running for figuring out pain points and trying to optimize their workflow

workflows using technology to support them.

Software engineers work often with site reliability engineers, or Sysadmins is maybe there, the

role could be called to actually maintain and help run the software and deploy it and

actually get it running somewhere that's not on your desktop, you often work with more

dedicated test engineers who are responsible for end to end testing, automated testing,

integration testing, at all, at all points of the lifecycle.

So as an engineer, you're never alone.

You're never working alone.

It's just a matter of like knowing how to communicate directly with the people on your

team

Hello, welcome to Overview of Web and Cloud Development.

When you’re starting out as a Web Developer, it can be difficult to determine what you

need to learn and what order you should learn it in.

Understanding how familiar websites are constructed and

delivered to you is a good starting point.

Let’s review the basics of how you interact with a website.

You launch an internet browser – there are lots available:

Google Chrome, Microsoft Edge, Mozilla Firefox, and Apple Safari are some of the most popular.

The browser has an address bar, into which you enter a URL, like www.IBM.com.

The browser then contacts the server with the name and requests the information that

makes up the website.

The server then sends a response, which contains the data that the client requires to display

the website.

For most websites, the server will return:

HTML, which defines the structure of the page, but doesn’t look very attractive

CSS, which adds style and flair to the page and JavaScript, which adds interactivity and

dynamic content.

Content displayed by websites can contain elements that are either previously stored

on the server (called “static”) or generated each time they are requested by the client

(called “dynamic”).

Dynamic elements can involve information coming from other systems and applications, such

as databases.

Most websites contain static and dynamic elements to provide the best user experience.

Cloud Applications are similar to Websites in that they request content that a server

returns.

Cloud Apps are built to work seamlessly with a Cloud-based back-end infrastructure, Cloud-based

data storage and data processing, and other Cloud services, making them very scalable

and very resilient.

The environment for building websites and Cloud Applications is divided into two primary

areas:

front-end and back-end.

The front-end deals with everything that happens at the client-side – everything the user

can see and interact with.

You can choose to specialize in front-end coding, using HTML, CSS, JavaScript and related

frameworks, libraries, and tools.

The back-end deals with everything that happens on the server before the code and data are

sent to the client.

The back-end coding usually handles the logic and functionality that make the website or

app work, and the authentication processes that keep data secure.

Back-end developers may also work with relational or noSQL databases, even collaborating with

database administrators in bigger projects.

Full-stack developers have skills, knowledge, and experience in both front-end and back-end

environments.

Whichever way you choose to specialize, you will need the appropriate tools to help you work.

The first tool most developers add to their resources is a code editor.

Developers also need tools to integrate, build, compile, and debug code.

Integrated Development Environments or IDEs incorporate some of these additional capabilities

beyond just code editing and make it easier to build and manage your code.

Good IDEs support multiple languages and integrate with management and storage tools like Git

and GitHub.

Other useful features are custom extensions and themes for supporting your working style

and environment.

Examples for code editors and IDEs include Sublime Text, Atom, Vim, VS Code, Visual Studio,

Eclipse and NetBeans.

Play video starting at :4:5 and follow transcript4:05

In this brief overview, you learned about:

The basic communication between client and servers.

How websites are built and displayed.

Front-end development is about what happens on the client side.

Back-end development is about what happens on the server.

Full-stack development incudes both front-end and back-end development.

IDEs will help you create and manage your code.

Internet websites offer lots of different services, one of the most popular being online

shopping.

When you explore an online shopping website, navigating through pages, choosing different

product categories, or comparing products, you are interacting with the front-end of

a website.

Let us see how the front-end of a website is developed by front-end developers.

For this, we need to understand how a website is made.

To create a website, web developers usually use Hypertext Markup Language (HTML), Cascading

Style Sheets (CSS) and JavaScript. These languages are designed to work in conjunction with each

other.

HTML is used to create the physical structure of a website.

The physical structure contains elements such as text, links, images/videos, page dividers

and buttons.

The HTML code ensures a proper formatting of all text and image elements so that browsers

display the page consistently.

The backend developer codes the structure of the website.

A website is like a house which has only been constructed.

Just like we need interior designers to add style to a space, we need front-end developers

to add the necessary glamor and appeal to a website.

When you order products from any website, you realize that the pages have a pleasing

font, attractive colors, and are easy to navigate.

Developers use CSS to create stylish websites.

CSS provides front-end developers with a standard method to define, apply, and manage different

sets of style characteristics for a website and each of its components.

CSS ensures uniformity in look and feel, style, colors, fonts, designs and layouts.

So, HTML is used to create the structure and CSS is used to design it and make it appealing.

CSS is also used to create websites that have cross browser compatibility which means that

they are compatible with multiple browsers and multiple devices such as PC, mobile devices,

iPads etc.

Online shopping websites are intuitive, interactive and quick to load. This is where

JavaScript comes into the picture.

JavaScript is an object-oriented programming language that is used in conjunction with

HTML and CSS to add interactivity to a website.

For example, you use HTML to add a login button to a page, and CSS to style that button.

You then use JavaScript to add log-in functionality to that button.

A new front-end development language is Syntactically Awesome Style Sheets called SASS.

It is an extension of CSS that is compatible with all versions of CSS.

SASS enables you to use things like variables, nested rules, and inline imports to keep things

organized.

SASS allows you to create style sheets faster and more easily.

Another language that is being used now is Learner Style Sheets or LESS.

LESS enhances CSS, adding more styles and functions. It is backwards compatible with

CSS.

Less.js is a JavaScript tool that converts the LESS styles to CSS styles.

Using all these languages, websites are designed as reactive and responsive.

Reactive or adaptive websites display the version of the website designed for a specific

screen size.

For example, a website can provide more information if opened on a PC than when opened on a mobile

device.

Responsive design of a website means that it will automatically resize to the device

it is being accessed from.

For example, if you open up a products website on your mobile device, it will adapt itself

to the small size of the screen and still show you all the features.

A JavaScript framework is an application framework that is written in JavaScript.

Programmers can manipulate the different functions, use them wherever required and can create

device responsive applications.

A few examples of several frameworks being used are:

Angular framework: an open-source framework being maintained by Google.

Angular frameworks allow websites to render the HTML pages quickly and efficiently.

It has built-in tools for routing and form validation.

React.js has been developed and maintained by Facebook.

It is a JavaScript library that builds and renders components for a web page.

It is not a complete suite of tools.

For example, routing is not a part of this framework and will need to be added using

a third-party tool.

React.js only helps build and drop components into a page.

Vue.js is maintained by the community and its main focus is the view layer which includes

user interface, buttons, and visual components.

It is flexible, scalable and integrates well with other frameworks.

It is very adaptable. It can be a library, or it can be the framework.

The task of a front-end developer evolves continuously.

The technologies are upgraded constantly and so front-end developers needs to keep

upgrading the websites that they create.

The websites that they create should work in multiple browsers, multiple operating systems

and multiple devices.

Welcome to the importance of back-end development.

A front-end developer creates websites

and Cloud applications using HTML,

CSS, and JavaScript to create what

the user sees and

interacts with in the client's software.

A back-end developer creates and

manages all the resources that are

needed to respond to

the requests that the user makes through the client.

The back-end developers tasks focus on enabling

this server infrastructure or

back-end to process requests,

supply data, and provide other services securely.

Front-end and back-end developers

must work together very closely.

Each needs to understand

the requirements of the solution and how

their respective parts will interact

before the development process can begin.

Throughout the life cycle of the website or Cloud app,

front-end and back-end developers collaborate

to resolve issues and add functionality.

How does the work of a back-end developer affect you as

you are browsing the Internet or using a Cloud App?

Think about it like this.

When you're shopping online,

what happens to the data you enter?

Your login information, your

product searches, your payment info.

The back-end processes all of these things.

The back-end developers write and maintain

the parts of the application that process the inputs.

Let's think about your experience as you explore

an online shopping site and make a purchase.

As you search for products,

your search request is submitted to a web application,

which then retrieves the data from

a separate database and

serves it back to the client for display.

To facilitate this, a back-end developer

must understand the language

that the web application uses,

how to query the database for the correct data,

and how to bring the two together.

Even a simple task like navigating around

the site can require the skills of a back-end developer.

Many sites have restricted areas that are only

available to users who have

an account and have logged in.

User account management, authentication and

authorization can be the responsibility

of the back-end developer too.

Once you've decided what you want to purchase,

you must add it to your card and make a payment.

The purchase process requires you to submit

sensitive information such as

your address and credit card number.

The back-end developer must ensure that

this data is securely handled and stored.

Front-end client interactions,

whether a request for data, like an image,

accepting input from a user filling out a form,

or securing sensitive information

like a credit card number,

all require different services from the backend server.

Each request needs to

interact with the back-end in a different way.

Back-end developers use APIs, routes,

and endpoints to process incoming requests.

An API is code that works with data,

usually using JSON or XML.

APIs have set rules and structure.

A route is a path to

a website or page that the user interacts with.

Routes generally take user input

and show results based on

the input and end point

maybe an API or may simply be a path.

When a request from the front-end

arrives at the back-end,

it is routed to the correct service.

If the back-end has an endpoint

defined for the request by using routing,

the request will be addressed and replied to.

If the endpoint is missing,

the server returns a 404 error.

Back-end developers must create

and maintain this server side routing.

Along with back-end APIs,

routes effectively allow the front-end client to

plug into the correct socket at the back-end.

APIs provide a mechanism for Cloud apps, mobile apps,

and other types of software to

access resources from the back-end.

To perform all this back-end development,

you will need to be familiar with

at least one back-end language

and its associated frameworks.

Among the most popular languages today is JavaScript,

which was originally designed to run in web browsers,

adding extra interactivity and

dynamic content to web pages.

JavaScript is also being used on

the back-end with

new releases adding server-side functionality.

JavaScript has many frameworks,

but two of the most well-known are Node.js and Express.

Python is another popular language.

It's very flexible and easy to learn.

Python has wide functionality.

It can be used for everything from creating

webpages to connecting to a database,

to performing data analysis.

Two well-known Python frameworks are Django and Flask.

Back-end developers often work with data and databases.

You will also benefit from learning some SQL.

To help handle requests from databases,

back-end developers can use

object relational mapping tools or

ORM to connect to

the databases and retrieve the correct data.

Although an ORM can hide some of

the complexity of querying databases,

it's useful to understand the fundamentals of

databases so that you can

troubleshoot any issues that arise.

The day-to-day tasks of a back-end developer focus

on the behind the scenes

functionality that keeps websites,

Cloud apps, and mobile apps up and running.

Back-end development covers a wide range of technologies,

from managing user accounts,

authentication and authorization to ensuring

that sensitive data is stored and transferred securely.

Back-end developers also work with databases,

retrieving, processing, and storing data as required.

Life for a back-end developer is

very challenging and ever-changing.

Welcome to Teamwork and Squads.

After watching this video, you will be able to define teamwork and describe the advantages

of collaboration in software engineering and describe squads.

By definition, a team is a group of people working together towards a common aim.

Within a team, you’ll find a range of different people with different skills, experience,

and talents.

Each person can give their attention and effort to the things that they are good at, and by

working alongside others on tasks outside their current repertoire, they can expand

their skillset.

Working in a team promotes creativity.

Collaborating with others gives you the opportunity to discuss ideas and challenge one another’s

thinking about a subject.

Good teamwork is empowering: positive attitudes and behaviors can impact the rest of the team

and create positive results.

Working well as a team doesn’t always come naturally, but there are some things you can

consider to help your team succeed.

Each member needs to trust and respect the other members of the team.

This generally comes with time, but depends on all of the members contributing equitably.

Defining and agreeing on goals for a project is essential so that the whole team knows

what they are working towards.

And you also need to define and agree on roles to avoid any duplication of effort or missed

tasks.

Working with each members strengths is important to make the most of the talent within your

team,

as is celebrating success and analyzing problems.

Communication is vital in a team environment.

Ensure that you choose a method that works for everyone so you know that the whole team

is seeing and responding to information.

So, what does teamwork look like in software engineering?

Teams often start projects with a kick-off meeting where they plan how they will complete

the project, assign tasks, and agree on goals.

Throughout the lifetime of a project, you’re likely to have whole team and/or sub team

meetings to review progress and plans.

Design and code reviews can be requested at the team level and undertaken by whoever has

availability at that point in time.

Team members might present walkthroughs of their sections of responsibility to the rest

of team so that the whole team has oversight of all parts of the project.

And, key team members will likely present walkthroughs to stakeholders at various times

during the project.

When a project is complete, retrospective meetings may be held to review what went well

and what could be improved in future projects.

You may have a mentor who may or may not be in your current project team.

You might also be asked to be a mentor.

Sometimes team mentoring is used so everyone can learn from each other.

Some groups also have teams working on internal projects such as defining code standards,

maintaining or updating legacy cross-project code, or reviewing potential new software

for usefulness to the team.

Good teamwork can bring many benefits to a project.

Working alongside others can encourage creativity and enable you to take advantage of each person’s

strengths while also allowing them to gain knowledge and skills from other members of

the team.

When working as part of a team, software engineers are more likely to adhere to corporate coding

standards and regularly document their code.

The additional accountability that teamwork creates results in better quality code, fewer

bugs, and more maintainable code.

From a software engineer’s point of view, working as a team can reduce stress because

there’s always someone to turn to and get help from.

And having someone to discuss problems with can help you to increase your understanding

and resolve more issues by yourself.

By working in a team, each member has a greater idea of the bigger picture, resulting in a

more coherent overall solution.

Some organizations that follow Agile development methodologies may call a team a squad.

Typically, a squad is a small team of up to 10 developers.

It is likely to consist of:

A squad leader who acts as the anchor developer and coach for the squad.

And a few software engineers who develop and implement the product features and test cases.

It may also include one or two user experience developers or designers.

In some squads the developers may work together in pairs to practice pair programming.

You’ll learn more about pair programming in another video.

In this video, you learned that:

Software engineering teams meet regularly throughout the duration of a project.

Good teamwork encourages creativity, shares knowledge, and results in better quality code.

Squads are small teams used in agile development.

Welcome to Insiders' Viewpoints: Teamwork in Software Engineering.

In this video, we will hear from experts discussing how teamwork is important in software development,

and how teams work together.

Yeah, so teamwork is very, very important in the world of software development.

As a software engineer, you're not just going to be sitting at your desk coding all day.

One of the most important aspects of working on a team or on an engineering team is communication,

and knowing how to collaborate and speak with your team members about the code you've written,

especially if you're going to a UX designer, and say, Hey, here's what I can do.

And your spec is a bit off.

Can we massage your your specs a bit just so we can meet the code coding requirements.

Incredibly important, I while I think it's possible for a single person to engineer and

deploy and run software, I, I think doing it as a team is both more impactful and gets

you more potential to do bigger and greater things, I think it's also just a more fulfilling

process, when you're when you're, you're building something with a group of individuals, I think,

you know, with without the support structure, it's hard to build something and support it

and maintain it just by yourself.

So teamwork, to me, is essential, because we're talking to each other all day.

And we're not only talking to our tech leads or senior engineer, we're talking to product

managers, we're talking to UX designers, we're talking to business analysts, we're talking

to data analysts, and so on and so forth.

You have the support system, you have diversity of opinions and perspectives in terms of how

you go about building it, it's more satisfying, you kind of can take larger risks, because

you're, you're in it together, and you're there to support each other, I think the celebration

of of a job well done is is a lot more, you know, it's a lot more fulfilling when you're

when you're doing that with someone else.

But I think it just gives you both kind of a larger range of scope and a larger sort

of risk portfolio, if you will, and you're able to, I think just, you know, get a lot

more done a lot more checks and balances and really gives you you know, less likely, there's

a less likely chance that you're gonna miss something, or you're gonna make a mistake,

there's a lot of a lot of support there.

And so every project I've worked on has had at least three other people that I have needed

to communicate with and support to be a reliable resource for them to go to them for questions.

Like I always have to be talking to somebody about something.

So teamwork is crucial.

Like, you can't, you can't work in an isolated silo forever, at some point, you have to talk

to someone else.

Building a project product takes a lot of people, a lot of us come together to make

a whole and to get the product shipped to production so our users can use it.

And so teamwork is imperative.

Welcome to Pair Programming. After watching this video, you will be able to describe pair programming and compare different

styles and list the benefits and challenges of pair programming.

Pair programming is an extension of teamwork where two developers work side-by-side at

one computer.

They can either be physically at the same computer or work virtually via video link

or shared screens.

The former is the preferred option, but virtual pair programming can also be productive.

Pair programming is a type of Agile development where two developers can plan and discuss

their ideas continually as they create a solution, generally resulting in a better end product.

There are various styles of pair programming: Driver/navigator style is the most common

style, where one developer is the driver, typing in the code, and the other is the navigator,

reviewing the code as it’s written and giving directions where to go next.

The navigator also keeps an eye on the bigger picture of the overall solution.

When working in this way, it’s important to regularly swap roles to keep both of the

pair engaged across the whole task.

Ping-pong style incorporates test-driven development.

For each task, one developer writes a failing test and then the second developer writes

code to pass that test.

For each new task, they swap roles, so regularly changing who writes the test and who writes

the implementation.

The two developers work together at the end of each task refactoring the successful code

to refine and improve it.

Strong style pair programming is a good way for junior software engineers to learn from

more experienced ones.

The defining rule for this is that for an idea to go from your head to the computer,

it must go through someone else’s hands.

So, the more experienced of the pair is the navigator and the driver learns from witnessing

their implementation and thought processes.

For this to work well, the driver shouldn’t challenge any ideas until the full implementation

is complete so as not to interrupt the flow of ideas from the navigator. Now

pair programming has many benefits.

It’s a good way to share knowledge and skills from one developer to another or between the

two and a great way for a new team member to get up to speed on a project.

As well as enhancing the technical skills of the pair, it’s also a good way of building

soft skills such as communication and problem solving.

Having two sets of eyes on the code often results in fewer typos, logic errors, and

bugs.

And it enables code reviews to be done on the fly.

While this doesn’t replace formal code reviews, it does enable another layer of review.

Having two people thinking about a problem can result in multiple initial ideas, but

is likely to result in the optimal approach being chosen earlier in the process.

And although pair programming can take longer than individuals writing the same solutions,

it’s likely to result in better code with less time spent reviewing, testing, and bug

fixing.

There are also a few challenges to overcome.

Working in a pair requires long periods of focus which can be exhausting for the two

programmers.

And personal or other work commitments can impact the pair schedule.

Sometimes one of the pair can end up controlling the entire process, resulting in a more

typist/programmer pairing which doesn’t benefit from any of the positives of pair

programming.

And at other times, individual personalities may not work well together.

When multiple sets of pair programmers are present, their discussions can result in a

noisy environment for the other workers in the room.

In this video, you learned that: Pair programming is an Agile development technique where two

developers work alongside each other.

There are multiple styles of pair programming.

And pair programming builds technical and soft skills, results in better quality code

and solutions, and increases overall efficiency.

Welcome to Insiders’ Viewpoints: Pair Programming.

In this video, software engineers will discuss their experiences with pair programming and

share their thoughts about the benefits and potential challenges associated with this

way of working.

First, we'll begin by asking about their experiences with pair programming and about the benefits

they associate with it.

At every job I've ever had, there have come times where you pair up with someone else,

you get someone else sitting next to you at a computer, you're sitting next to them at

their computer, you're looking at a problem together, you're thinking about it.

Yes, I have actually, some of the benefits are definitely working with an engineer who

might be more of a senior engineer, because they have a lot of insight.

And they obviously have a lot of experience doing what you're probably trying to figure

out how to do.

And it's really great when it's someone who can both guide you but also kind of take take

a step back and not take over at the same time.

I think pair programming can can be very beneficial.

I think.

If you're when you're working with someone together, it's an opportunity to learn I think

pair programming can be very great in circumstances where someone is new to a team.

New to a programming language new to a set of tools, or a development environment

that is new to them, sitting next to someone who's an expert, I think can both help

you ramp up more quickly, it can help the the team itself, sort of understand where

you are, in terms of your own knowledge and journey and help put you in a place to

both succeed and utilize you in your skills as best as possible.

It does help you get real time feedback, you don't necessarily have to wait for code

to be sent out to be reviewed, and you can kind of, you know, brainstorm together

in real time on the actual implementation of it.

Ultimately, I try to keep in mind that the goal of pair programming is that everyone

has inherent blind spots in the way that they think.

And that the way that they approach a problem.

And if you are talking to somebody else, they can point out what your blind spots are.

And they're going to have different blind spots than you as well.

So you can point out their blind spots.

What do you think are the drawbacks and challenges of pair programming?

I think that can be a con sometimes when you're working with another engineer and you're

trying to pair program with someone who kind of like takes over and then doesn't allow

you to do your part or to learn especially if they kind of like already know how to do

something.

So it's kind of like a double-edged sword.

On the one hand, you want to get that practical experience of practice of pair programming

and you want to get that experience from someone else.

But at the same time, you want to make sure that you're learning.

I personally find pair programming to be a little frustrating.

Because the ways I go about solving problems can be very different than the ways other

people go about solving problems.

And it's not that I have some special, unique, wonderful technique, it's just that everyone's

brain works differently.

And watching someone else go through their workflow of how to solve a problem, they're

not clicking the exact same buttons that you are.

And that can get very frustrating to me.

It's like, why are you using your mouse because all those things, there's keystrokes

or like, you could be doing this so much faster.

And so I find it a challenge to do pair programming sometimes.

You actually need calendar time.

So you need to be able to dedicate time to sit together and work together and have an

aligned schedule.

You both you can either be physical, could do it virtually, but you need to make the

time to do it.

Pair programming, some can come with some overhead in terms of getting

off the ground.

So if you have a tight time crunch, competing priorities, maybe just not enough, there might

be not enough time in the schedule to allow for it, that can be one of the cons.

That it can while you can get long term, I think efficiency benefits from you know,

built, did writing the code together, because then support in the future will be easier

because you have more people who know how the code was written and what the intent was.

So there are definitely long term benefits.

But there could be you know, some short term costs to it.

So there are trade offs in that sense.

Welcome to Introducing Application Development Tools.

Getting your Cloud App from the ideas stage to fully formed, written, and deployed is

a long process, but there are many tools which will help you along the way.

A cloud application developer’s workbench includes:

Version Control, Libraries, and Frameworks.

When many developers are working on the same project, knowing what order changes were made,

thereby creating a new version of the source code, becomes overridingly important.

Play video starting at ::40 and follow transcript0:40

Version control systems keep track of what changes were made when and by whom and resolve

any conflicts between changes.

For developing your code version control can be useful even when you are the sole contributor

on a project.

Properly used, it can give you a way to revert to an older version of your code if something

goes wrong and gives you some basic information about how the code developed over time.

Version control functionality is generally tied to the storage system you are using,

which is why a code repository is recommended, even for beginners.

Git and GitHub are extremely popular for source code storage and management.

Git stores files in repositories where you can track changes, split code into different

branches for more focused development, and then merge them back into the main body of

code.

Play video starting at :1:36 and follow transcript1:36

Libraries are collections of code, like standard programs and subroutines, that you can use

within your code.

For example, you might want to include a navigational feature, like a carousel – a code library

can supply you with the code for that so that you don’t have to spend the time and energy

creating one from scratch.

Being able to reuse code in this way makes developing your app much quicker and easier.

Multiple code libraries can be integrated into your existing project.

As you discover a need for a specific function or feature, you can research an appropriate

library.

You determine when to call the required method as needed. The control returns to the program

flow once the subroutine is finished.

When you use a code library, you are in control.

Code libraries are generally used to solve a specific problem or add a specific feature

set.

Either way, there are lots for you to choose from, so do your research.

Play video starting at :2:41 and follow transcript2:41

Here are some examples of code libraries:

jQuery is a JavaScript library that simplifies DOM manipulation.

Email-validator is a small library that checks an email address is correctly constructed

and valid.

Apache Commons Proper is a repository of reusable Java components.

Play video starting at :3:3 and follow transcript3:03

Frameworks provide a standard way to build and deploy applications.

You can think of a framework as being a skeleton that you can extend by adding your own code,

providing a scaffold on which to build your apps.

The framework you intend to use must be determined early in your development planning and used

right from the beginning.

New frameworks can’t be incorporated into an existing project.

Your chosen framework dictates the architecture of your program and controls the program flow.

The framework determines which subroutines and methods will be called when. When working

with a framework, there is a specific structure that you must follow.

The framework calls on your code, rather than you calling on the framework.

Frameworks are less flexible than libraries, allowing you less control, but they do provide

good standardization and can help you create efficient code.

To use an analogy, if you are a carpenter building a house, the framework is the frame

that you add to – bricks on the outside, plasterboard on the inside, and so on.

The frame acts as a guide for how the house is constructed.

Here are some examples of frameworks:

AngularJS is a JavaScript-based framework for dynamic web applications.

Vue.js is a JavaScript framework focused on the user interface.

Django is a framework that uses Python for web development.

Frameworks define the workflow that you must follow, unlike libraries, which allow you

to call functions as and when required.

When using a framework, it can sometimes feel like you, as a developer, are not in full

control of the development process.

This sense of the framework and its predefined workflow controlling the development process

is referred to as inversion of control.

Frameworks that have a lot of control are known as opinionated – they have opinions

on how their workflow should be used and remove a lot of the decisions you would otherwise

have to make about how code is written, the location of files, and even file names.

Frameworks often include their own libraries, which they call when needed.

Inversion of control allows you to create standardized apps, and takes away a lot of

the tedious configuration work, so you can focus on the code for your app.

In this video, you’ve learned about some of the tools that you will utilize in your

career as a developer including:

Version control, libraries, and frameworks.

Welcome to More Application Development Tools.

Let’s look at some tools which can help you get your app built and deployed:

CI/CD, Build Tools, Packages, and Package Managers.

Play video starting at ::21 and follow transcript0:21

CI/CD refers to the practices of continuous integration and either continuous delivery

or continuous deployment.

CI/CD is a best practice for devops teams enabling developers to deliver frequent changes

reliably.

Implemented through a build-automation server, Continuous Integration (CI) ensures that all

the code components work together smoothly.

A CI build environment enables you to integrate newly developed code frequently, at least

every day, if not every hour, depending on how quickly the project changes.

Continuous delivery (CD) begins where CI ends. The CI process automatically builds and tests

your code, then CD deploys all code changes in a build to a testing or staging environment.

A build tool transforms your source code into the binaries needed for installation.

Build tools organize your source code, set compile flags, and manage dependencies.

They are most important in environments where there are many inter-connected projects, with

multiple developers contributing to each project.

In these environments, it can be very difficult to keep track of what changes were made, in

what order, what dependencies exist, and what needs to be incorporated in the next build,

so automation is key to keeping everything running smoothly.

Play video starting at :1:50 and follow transcript1:50

Build automation can automate a wide variety of tasks that developers do in their day-to-day

activities like:

Downloading dependencies.

Compiling source code into binary code.

Packaging that binary code.

Running tests.

And deployment to production systems.

Play video starting at :2:9 and follow transcript2:09

You can initiate a build from the command line or from an IDE.

There are two categories of Build Tools widely in use:

Build-automation utilities, which generate build artifacts like executables, by compiling

and linking source code.

Build-automation servers, which execute build-automation

utilities on a scheduled or triggered basis.

Play video starting at :2:33 and follow transcript2:33

Some examples of build tools are:

Webpack – a module bundler for JavaScript.

Babel – a JavaScript compiler.

And Web Assembly - a binary instruction format that runs in your browser.

Play video starting at :2:48 and follow transcript2:48

Now that you have your app developed and tested, you’re ready to deploy.

But how does that happen?

The app needs to be simple and trouble-free for the user to install, so a commonly used

technique is to collect all the necessary files and bundle them together into a package.

Play video starting at :3:7 and follow transcript3:07

Packages are archive files that contain the app files, instructions for installation,

and any metadata that you choose.

They have their own metadata too, including the package description, package version,

and any dependencies, like other packages that need to be installed beforehand.

Play video starting at :3:27 and follow transcript3:27

Once you have bundled your app into a package, you can use a package manager to distribute it.

Package managers take care of the tasks of finding, installing, maintaining or uninstalling.

software packages at the user's request.

Play video starting at :3:42 and follow transcript3:42

Package management systems:

Coordinate with file archivers to extract package archives.

Verify checksums and digital certificates to ensure the integrity and authenticity of

the package.

Locate, download, install, or update existing software from a software repository.

And manage dependencies to ensure a package is installed with all packages it requires.

Some commonly used package managers for each of the major platforms are listed here:

On Linux - Debian Package Management System (DPKG). Red Hat Package Manager (RPM).

On Windows - Chocolatey.

On Android - Package Manager.

On MacOS - Homebrew and MacPorts.

Play video starting at :4:31 and follow transcript4:31

Any libraries or utility code that is developed as part of the application

is managed with the cloud application package managers.

Here are some examples of package managers for popular languages:

For Node.js/Javascript - npm.

For Java - Gradle and Maven.

For Ruby - RubyGems.

For Python - Pip and Conda.

In this video you’ve learned about some of the tools that you will utilize

in your career as a developer , including:

CI/CD, Build Tools, Packages, and Package Managers.

Welcome to Introduction to Software Stacks. After watching this video, you will be able to

describe software stack terminology,

identify some commonly used software stacks,

and list the benefits and challenges of different software stacks.

Play video starting at ::22 and follow transcript0:22

A software stack is a combination of technologies that includes software and

programming languages. Developers use a software stack to create applications and solutions such as

web and mobile apps. The set of individual technologies is stacked in a hierarchy and

work together to support the execution of an application. The higher levels in the stack

provide tasks or services for the user and the lower levels interact with the computer hardware.

Software stacks typically include:

Front-end technologies such as programming languages, frameworks, and user interface tools.

And back-end technologies such as programming languages,

frameworks, web servers, app servers, operating systems, messaging applications, and databases.

You might hear the term technology stack used in place of software stack. However,

a technology stack is a broader term that includes hardware and infrastructure like

virtual machines, containers, storage, and load balancers, as well as the software stack.

The simplest implementation of a software stack consists of a presentation layer,

a business logic layer, and a data layer. However, more complex applications use more complex stacks,

which could include software for virtualization, scheduling and

orchestration, runtime environments, database connectivity, networking, and security.

The software and services that make up a stack can be from a variety of sources:

from internal resources, to third party providers, to cloud providers.

There’s no formal definition of the structure of a stack, the only rule being that the software and

services included must support an application’s development, functionality, or deployment.

When you’re using a software stack, you don’t have to use all of the available layers,

you only need to use those which are relevant to your solution.

There are many different examples of software stacks,

The Python-Django stack, uses the popular Python programming language alongside the

Django web framework. This combination is all open source and commonly used for large-scale,

fast-changing web applications.

The Ruby on Rails stack, uses the Ruby programming language with a server-side

web application framework. Ruby on Rails is great with JSON or XML for data transfer

and HTML, CSS, and JavaScript for front-end development.

And the ASP.NET stack includes Microsoft technologies such as the ASP.NET MVC

framework, the IIS web server, SQL Server, and Azure.

You’ll learn more about the LAMP, MEAN, MEVN, and MERN stacks next.

The LAMP stack runs on the Linux operating system. It uses the Apache HTTP or Web server,

MySQL databases, and the PHP programming language. LAMP is an example of an early

incarnation of a software stack designed for building websites and cloud applications.

All its constituent parts are open source and loosely coupled, so it’s easy to swap

different options into the stack. For example, you could choose to use PostgreSQL instead of

MySQL for your database server, changing the LAMP stack to be the LAPP stack.

Similarly, you could use the Python programming language instead of PHP.

The MEAN stack uses a MongoDB database with an Express.js web application server framework,

the Angular.js framework for front-end JavaScript development,

and the Node.js platform for server-side scripting.

The MEAN software and services are platform agnostic, free, and open source.

Play video starting at :4:23 and follow transcript4:23

There are other stacks related to the MEAN stack, including:

The MERN stack which replaces Angular.js with React,

and is a flexible and high-performing framework for developing front-ends.

And the MEVN stack which replaces Angular.js with Vue.js. Vue is a lighter-weight JavaScript

framework with less capabilities, but it can provide better performance than Angular.js.

Let’s consider some advantages and disadvantages of three different software stacks:

MEAN, MEVN, and LAMP.

MEAN is a free and open-source JavaScript software stack used for building web applications. The

biggest advantage of the MEAN stack is that all of the parts use JavaScript, so developers only

need to know a single language. The stack is also open source which means the cost is lucrative to

businesses and there is a lot of documentation and re-usable code for developers to use. Development

can happen quickly because Node.js has a huge collection of free, reusable module libraries.

However, the MEAN stack may not be well-suited for large-scale applications.

When using Express.js, the business logic often resides on the server preventing the reuse

of some services like batching operations. And MongoDB is great for unstructured data,

but it doesn’t provide the same level of functionality as a relational database.

The MEVN stack is a web stack like MEAN,

but it uses Vue.js instead of Angular.js for user interfaces.

MEVN and MEAN stacks have similar advantages,

but Vue.js is a much newer technology and doesn't have as many reusable libraries as Angular.js.

Like MEAN and MEVN, the software and services in the LAMP stack are open source meaning there are

lots of reusable chunks of code available to the developers. And because LAMP is one of the oldest

software stacks it’s easy to find support and reusable solutions. However, because the Linux

operating system is an integral part of the stack, it isn’t as flexible as MEAN and MEVN

which are platform agnostic. MySQL is a relational database, so the stack cannot take advantage of

unstructured data. The other disadvantage of the LAMP stack is that the back-end runs on PHP, Perl,

or Python, whereas the front-end uses JavaScript and HTML, making it harder for developers to

switch back and forth than the MEAN and MEVN stacks which use JavaScript throughout.

Play video starting at :7:11 and follow transcript7:11

In this video, you learned that:

Software stacks are a combination of technologies for creating applications and solutions.

Software stacks can range from simple three-layered systems to many layers.

There are numerous types of software stacks for different developers and environments.

The biggest advantage of the MEAN stack is that all of the parts use JavaScript,

so developers only need to know a single language.

And, the LAMP stack on Linux works well for relational data.

Welcome to Insiders’ Viewpoints: Tools and Technologies. In this video, we will hear

from experts discussing their favorite tools and technologies for software engineering projects.

Play video starting at ::18 and follow transcript0:18

Our team uses Git and GitHub every single day for all of our projects.

We use it for tracking code. We use it for collaboration and we use it for tracking bugs,

features, and tasks for all of our projects.

I believe most open source projects are now using Git. Every project I've contributed to,

whether open source or closed has used it Git's true value is most apparent when you're working

on a team with multiple people, at which point you'll be glad to have things like

feature branches and pull requests. But even if you are working on a project where you're

the sole committer, I recommend using Git, and in particular GitHub for the Community aspect

So if you are a front end developer, if you understand JavaScript and picking up

backend using a framework, like Node JS is not that much of a stretch, I actually started building,

Play video starting at :1:17 and follow transcript1:17

you know, static websites, and then slowly learned how to how to put in dynamic content

using JavaScript. And then I had to use Java, or PHP to build my back-end systems in on the server.

For front end development, you're primarily using HTML, CSS, and JavaScript. If you're very

front-end focused, you might want to try out an IDE built for that specific purpose,

such as brackets if you want to keep things more general, I recommend VS Code.

In either case, I recommend installing IDE extensions for automatic formatting and linting

of your code. By using prettier and eslint these will help you catch issues as early as possible

Play video starting at :1:59 and follow transcript1:59

for back end development the tools are quite a bit more varied so it's

hard to give specific recommendations. But if you choose Node.js, you'll be able to use some

of these same tools that I mentioned for front end development. Otherwise, I recommend just

looking into what IDE extensions are available and what linters and auto formatters are available

Play video starting at :2:25 and follow transcript2:25

JavaScript is a very powerful language. It lets you do pretty much anything you can imagine,

including shooting yourself in the foot if you're not careful.

The practices I typically follow with JavaScript include proper scoping of variables and functions.

And writing unit and integration tests for my features.

Play video starting at :2:44 and follow transcript2:44

React JS was developed at Facebook very popular. Angular is a framework operated

by Google that allows for development of single page applications. SPA's

Play video starting at :3: and follow transcript3:00

jQuery is possibly the most popular and oldest library out there, created by John Resig in 2006,

and jQuery is frequently used with both Angular and React. Backbone is a lightweight JavaScript

library that is very popular in terms of back-end languages and frameworks.

Node.js is an open source server side platform built on the Google Chrome JavaScript engine.

It uses an asynchronous single threaded architecture that allows it to serve a

very large number of concurrent connections. Flask is a framework used on Python that is popular with

Pythonistas and the Spring framework based on Java has been around for years and remains popular

Play video starting at :3:54 and follow transcript3:54

We use React JavaScript framework it is better than Angular in terms of speed and efficiency.

React is easier to learn than other JavaScript frameworks, which makes its adoption in the

team easy. It also resolves, it also resolves the problems related to cross browser issues.

Another great feature of React is the use of JSX. JSX is helpful when working with UI

inside the JavaScript code. It helps react to show more useful error and warning messages.

My favorite front end JavaScript framework would be React JS, I love the whole component driven

Play video starting at :4:35 and follow transcript4:35

design and architecture that you have then that you have to follow when you create

a React JS based application. I also like the idea of, you know,

props and states to hold your, the state of your application.

By using Express JS for back-end development, we are able to scale applications quickly.

We can code both front-end and back-end with the help of JavaScript easily by using

Express JS. Express JS is supported with the Google V8 engine with the help of which

you will be able to get higher performance without any lag or error in processing.

Play video starting at :5:13 and follow transcript5:13

It also supports the caching features, so you would not have to re-execute the

codes again and again. Moreover, moreover, it will help web pages to load faster than ever

When working with Node.js, I'll, I'll give you two of my favorite

packages that I use on a daily basis. So first is making requests to web services.

I use a library like Axios, that helps me create these requests with the right headers, and also

provide me with callback functions and or promises to be able to handle the responses that come back.

Play video starting at :5:50 and follow transcript5:50

My second package would be when I'm working with databases,

I'll use NPM packages, more likely than not to talk to an external database,

regardless of if it's, if it's a relational database, or a no SQL database.

So in writing JavaScript, I really like to take advantage of the features and ESX, because

it can make my code look cleaner, easier to read, and, and just more beautiful,

Play video starting at :6:21 and follow transcript6:21

such as the arrow functions, or the dot dot dot operator. And so I encourage you to,

after learning JavaScript, dive into ESX a bit and have fun with it.

***MODULE 3***

Welcome to interpreted and compiled programming languages!

After watching this video, you will be able to:

Identify interpreted programming languages,

and identify compiled programming languages.

What are programming languages?

Programming languages help us tell computers what to do.

Computers don’t use human language; they use their own language, called machine code.

Machines understand binary code, that is 1s and 0s.

So, to make communicating with computers easier, we have human-readable programming languages.

Two common categories for programming languages are:

Interpreted

and compiled.

These are broad categories, and there are many programming languages that are classified

under compiled languages and interpreted languages.

The purpose of the project helps a developer determine which programming language is best.

Interpreted language is also commonly referred to as scripted or scripting language.

Programs written in interpreted or scripted language, like Python and HTML, run through

the programming interpreter on your computer’s operating system or in your web browser.

Remember that programming languages are in human-readable code,

so, the interpreter takes the human-readable scripted code and then translates it into

machine code, enabling the computer to complete the requested task.

As computers and web pages have changed and advanced, some interpreted programming languages

are outdated, and they are not as useful or relevant

Some other languages are more versatile and easier to learn, which means they are preferred

over other scripting languages and used more often.

All interpreted programming languages need an interpreter to translate the source code.

Translators are built into your web browser or they require a program on your computer

to translate the code.

Several different interpreted programming languages exist.

Some examples of common interpreted programming language types are:

• JavaScript, a simpler scripting language that runs through the web browser interpreter

• Python, a language that is popular because it is easy to learn and use for developers

• Lua, a general purpose, lightweight game scripting language that is easy to learn and

use

• HTML, a markup language used for formatting web pages

Another category of programming languages is the compiled programming languages.

Compiled programs are applications and programs, like your music app or your operating system,

that you run on your computer or device.

The programs are packaged --or compiled-- into one executable file.

They are usually larger programs.

Compiled programs are used to help solve more challenging problems, like interpreting source

code.

Compiled programming languages are often referred to more simply as programming languages.

A compiler program creates a program file, which runs the software.

Simply put: It piles the code into one file that runs when you double-click on the app

on your device.

The program runs faster and it can be done repeatedly.

The source code is converted from the programming language to machine code.

Then it is compiled into one executable file.

Finally, the program runs when you select the icon or file on your device.

Some examples of compiled programming languages are:

C and its variations C++ and C#, which are used in many operating systems, like Microsoft

Windows, Apple’s macOS, and the open-source operating system Linux.

Java is another compiled programming language.

It shouldn’t be confused with the interpreted language JavaScript.

The Android OS is written in Java because it works well across computing platforms.

Let’s look at this process:

When you update to a new version of your operating system, like Microsoft Windows, your device

might download an installation program.

That program is made up of many files.

The files are written in a compiled programming language.

These files give instructions to your device in machine code.

The compiled program is running on your device.

A compiled program that you commonly use is your device’s operating system, such as

Linux, Microsoft Windows, Apple’s macOS, or Android.

Your operating system is written in a compiled programming language, like C, C++, C#, or

Java.

In this video, you learned that:

Interpreted programming languages run scripts that are repetitive and need to be run often.

Interpreted programming languages are more versatile and can be used across platforms

as long as there is the correct interpreter.

Some examples of interpreted programming languages are JavaScript, Python, and HTML.

Compiled programming languages are for more complex programs that complete larger tasks.

Compiled programming languages are used for creating executable files that can run directly

from your device.

And some examples of compiled programming languages are C and Java.

MUSIC]

Welcome to comparing compiled and interpreted programming languages.

After watching this video, you will be able compare interpreted and

compiled programming languages and discuss interpreted and

compiled programming languages.

How do developers choose a programming language?

Developers choose which programming language is best to use depending on what

they have the most experience with and what they trust, what is best for

their users, and what is the most efficient to use.

What are interpreted and compiled programming languages?

Interpreted programming languages are used to create a scripted source code for

smaller tasks.

The source code goes through an interpreter.

The interpreter is built into the operating system on a computer or

on a web browser.

Compiled programming languages are used to create files.

The files are executable files.

They are then grouped in programs that you can run on a computer or device.

Interpreted programming languages are also called script code or

scripting, because they are used to automate tasks.

Interpreter programs read and

execute the source code line by line like someone would read a script.

Each time the program runs,

the source code needs to be executed to receive the desired output, and

source code written in one of the interpreted programming languages runs on

almost any operating system with the right interpreter.

For example, imagine you have a client requesting information about webpage

views for last month.

Which type of programming language would be the most appropriate and

efficient to create a program for this task?

You would use one of the interpreted programming languages to write code for

a script.

The script would retrieve the webpage views data and

then put it into a table for the client to read.

Compiled programming languages are also called programming languages for short.

They are used for more complex programs that complete larger tasks,

like running a spreadsheet program on your computer.

A compiled programming language is used to write a larger program,

usually installed on your device as an executable file.

Writing code in a compiled language does take longer, but

the payoff is that the programs code runs faster,

because compiled programs are installed on the device.

And once the program is coded,

the compiled program is grouped into one downloadable file.

Remember that client who wanted information about webpage views?

Now, they want a spreadsheet program to view and manage the data.

A programmer could use the C programming language to create a compiled

program like Microsoft Excel.

So what are the major differences between interpreted programming languages and

compiled programming languages?

First, there's interpreted programming.

These programs are available across multiple platforms or in the cloud.

The scripting languages are easier to learn and use, and

they're better for websites because they tackle smaller, repeated processes.

Then there's compiled programming.

These programs are available to users with the same operating system coded in

the same language.

However, compiled programming languages are more difficult to learn and use,

because they are larger programs with more parts, and they're better for

larger tasks, like running the operating system on your computer.

To better understand more of the similarities and

differences of programming languages, let's take a look at some examples.

A few of the most common programming languages are C, C++ and C#.

C is a compiled programming language.

C is the original language, and C++ and C# are variations.

C and its variations are case-sensitive languages.

The C programming language is the basis for Windows and

other common operating systems, and it takes more time to learn and

use for coding but requires less memory and the code runs faster.

Another common programming language is Java.

It is a compiled programming language.

It is also a case-sensitive, object-oriented programming language.

The Java virtual machine or JVM is required to run the code written in Java.

It is the primary programming language for some operating systems,

like the Android OS.

And a benefit of Java is that it is a cross-platform language,

which means it runs the same code on macOS, Windows, and Linux.

Python is an example of an interpreted programming language.

It is also referred to as a scripting language.

Python is a popular general-use, case-sensitive programming language.

It is used with Windows, macOS, and Linux operating systems, and with

server-side web app code, and it requires the Python engine to interpret code.

JavaScript is another interpreted programming language.

It is a scripting language that runs on the client side in web browsers.

JavaScript is case sensitive.

Simple scripts are run with HTML.

Complex scripts are run in separate files.

And while it sounds similar, it shouldn't be confused with Java,

the compiled programming language.

Another example of a common programming language is HTML.

It is an interpreted programming language.

HTML stands for Hypertext Markup Language.

HTML is mostly case insensitive with some exceptions, and

it uses tags to format webpages on client-side web browsers.

In this video,

you learned that interpreted programming languages create source code that runs

through an interpreter in your device's operating system or on your web browser.

Compiled programming languages create executable files that are grouped in

programs on your device.

Compiled programming languages like C and Java are used to write larger programs,

like operating systems and other executable files.

And interpreted programming languages like Python and HTML are used to write

code that can complete repetitive tasks within a web browser or a computer.

[MUSIC]

Welcome to “Query and Assembly Programming Languages.”

After watching this video, you will be able to:

Compare high-level and low-level programming languages.

Describe query languages,

and describe assembly languages.

Play video starting at ::24 and follow transcript0:24

Let’s explore query and assembly programming languages.

But first, we need to categorize programming languages

into two levels – high-level and low-level.

A high-level programming language is

more sophisticated and

uses the common English language to make its code more understandable

and to increase the speed of coding and debugging programs.

Examples of high-level programming languages include query languages

such as Structured Query Language (or SQL), structured programming languages such as Pascal,

and object-oriented programming languages such as Python.

In contrast, a low-level programming language

uses a set of symbols to represent machine code.

And examples of low-level programming languages include assembly languages

such as ARM, MIPS, and X86.

A query is a request for information from a database.

The database handles the query and searches its tables for the information requested

and returns the results to the querying entity.

When querying a database, it is important that both the user application making the query

and the database handling the query are speaking the same language.

In programming terms, writing a query means using predefined

and understandable instructions to make the request to a database.

This is achieved using programmatic code and this is what we refer to as a query language.

A query language may also be referred to as a database query language.

By far the most prevalent query language for database queries and database management is SQL.

However, there are other query languages available

such as AQL, CQL, Datalog, and DMX.

In addition to SQL databases,

there is another type of database called NoSQL, which stands for Not Only SQL.

The key difference between these two types of databases is their data structures.

While SQL databases are relational

and use structured, predefined, schemas,

NoSQL databases are non-relational in nature

and have dynamic schemas for unstructured data.

A query language is predominantly used to request data from a database

or to create, read, update, and delete data in a database. You will likely see the term

CRUD used to refer to these last four key database operations.

Typically, a user enters a command to either make a query or perform a CRUD operation using syntax

that is understandable to the database management system hosting the database.

And a database typically consists of structured tables made up

of multiple rows and columns of data.

When a user performs a query,

the database retrieves the data from the relevant rows and columns in the table and

arranges it into some sort of order,

ready to be returned and presented in the query results.

Database queries are either

a select command,

or an action command, such as CREATE, INSERT, UPDATE, or a mixture of both.

The term “statement” is more commonly used to describe these commands.

Select queries request data from a database,

whereas action queries manipulate data in a database.

Query statements can also be used to perform other administrative functions

such as creating users and modifying permissions.

This table lists some of the most common SQL query statements.

Play video starting at :4:5 and follow transcript4:05

Here are some simple syntax examples of common SQL statements.

Play video starting at :4:12 and follow transcript4:12

As mentioned earlier, assembly languages are less sophisticated than query languages,

structured programming languages, and object-oriented programming languages.

As an assembly language uses a simple set of symbols to represent the 0s and 1s

of machine code, it is categorized as a low-level programming language.

Assembly languages are closely tied to the processor architecture from hardware manufacturers

and therefore, each CPU type will typically have its own assembly language.

For this reason, there are a large number of assembly languages in use today,

which vary among hardware manufacturers.

Assembly languages use a simple readable format for their statements,

and they are entered one line at a time,

with one statement per line.

Assembly language statements use the standard format shown here.

In this syntax all fields in curly brackets { } are optional,

and the statement has two main parts.

The first part is the instruction (or the mnemonic),

and the second part includes the parameters (or the operands).

There may also be optional useful comments added on the end of the statement.

One other key difference with assembly languages

is that they are translated using an assembler instead of a compiler or interpreter,

and one statement translates into just one machine code instruction,

as opposed to high-level languages where one statement can be

translated into multiple machine code instructions.

Assemblers translate assembly language into machine code using mnemonics

such as Input (INP), Output (OUT), Load (LDA), Store (STA), and Add (ADD).

The statements consist of opcodes that tell the processor what to do with the data,

and operands that tell the processor where to find the data.

Play video starting at :6:13 and follow transcript6:13

In this video, you learned that:

Query languages, structured programming languages,

and object-oriented programming languages are categorized as high-level programming languages.

Assembly languages are categorized as low-level programming languages.

A query language is predominantly used to request

data from a database or to manipulate data in a database.

The most prevalent query language for database queries and database

management is Structured Query Language (SQL).

Select queries request data from a database,

whereas action queries manipulate data in a database.

You also learned that:

Assembly languages use a simple set of symbols to represent the 0s and 1s of machine code.

Assembly languages are closely tied

to the processor architecture from hardware manufacturers.

Assembly languages are translated using an assembler instead of a compiler or interpreter.

And assembly language instructions

have a one-to-one association with their machine code counterpart.

Welcome to Understanding Code Organization Methods.

After watching this video, you will be able to:

Explain why code organization methods are important when programming.

Define the different code organization methods

and identify the benefits of using code organization methods.

Organizing is very important when it comes to reading, maintaining, and configuring code.

Well-planned-out software design, usually using one of several methods, helps programmers

write cleaner and more reliable code.

Planning out code in a visual format helps improve the code base once it is written and

reduces the chance of bugs and errors throughout the lifespan of a project.

Organizing code before programming has a positive impact on the quality of the program and helps

provide a consistent and logical format to use while coding.

There are two main methods of organizing code: flowcharts and pseudocode.

The main difference between pseudocode and a flowchart is that the former is a basic,

high-level description of an algorithm.

An algorithm is a step-by-step sequence of solving a given problem.

A flowchart is a pictorial representation of an algorithm showing the steps as boxes

of various shapes and colors connected by arrows that indicate their order.

Flowcharts are used in designing or documenting a process or program.

Pseudocode provides a beneficial bridge to the project code because it closely follows

the logic that the code will.

Pseudocode also helps programmers share ideas without spending too much time creating code,

and it provides a structure that is not dependent on any one programming language.

Flowcharts are especially beneficial for smaller concepts and problems, while pseudocode is

more efficient for larger programming projects.

And flowcharts provide an easy method of communication about the logic and offer a good starting

point for the project because they are easier to create than pseudocode in the beginning

stages.

A flowchart is the graphical or pictorial representation of an algorithm using different

symbols, shapes, and arrows in different sizes and colors to demonstrate a process or a program.

The main purpose of using a flowchart is to analyze different methods of solving a problem

or completing a process.

Several standard symbols are applied in a flowchart, and you can easily highlight certain

elements and the relationships between each part in the process.

Some traditional flowchart shapes used for programming concepts are:

Start/End (a capsule),

Process (a rectangle),

Decision (a diamond),

Data (a parallelogram),

And Connecters (as arrows).

A simple flowchart to represent how to add two numbers is easy to create.

The procedure starts with the capsule, and has a connector, or arrow, progressing to

the next step, the data input (the parallelogram), in this case, input n1 and input n2.

Then, another connector, or arrow, goes to the process (the rectangle) of adding the

two inputs, Sum = n1 + n2.

Another connector, or arrow, moves to the data (the parallelogram) showing the results,

in this case, Print Sum.

The last connector, or arrow, moves to the end of the process, the end capsule.

Flowchart software is an application that provides various functionalities to create

flowcharts by providing the ability to drag shapes into the desired order using an easy-to-use

editor.

Flowchart software also provides team collaboration for creating flowcharts.

Some of the well-known flowchart software programs include:

Microsoft Visio,

Lucidchart,

Draw.io,

And DrawAnywhere.

Pseudocode is an informal type of programming description that does not require any strict

programming language syntax or underlying technology considerations.

System designers write pseudocode to ensure that programmers understand a software project's

requirements and align code accordingly.

Pseudocode is used for creating an outline or a rough draft of a program that summarizes

a program’s flow but excludes underlying details.

Pseudocode acts as the bridge between the programmer’s brain and the computer’s

code executor that provides the ability to plan instructions that follow a logical pattern,

without including all the technical details.

Pseudocode is a great way of getting started with software programming as a beginner without

worrying about coding syntax.

Pseudocode helps both programmers and non-programmers agree about the program’s goal and the basics

of how the task should be done.

Here’s one example of pseudocode written to check if the user entered an odd or even

number:

The next example displays the pseudocode written in C++:

There are many advantages of using pseudocode over flowcharts.

The main benefit of pseudocode is that it’s simple and explains exactly what each line

of an application should do.

The coder can focus more on logic than on program language syntax.

Removing the distractions of coding when using pseudocode makes the code development stage

easier.

Words and phrases in pseudocode represent lines of basic computer operations that simplify

translation from the pseudocode algorithm to the specific programming language.

Pseudocode allows programmers working in different computer languages to interact with each other.

Pseudocode can be reviewed by different development groups easier than real code.

Pseudocode is easier for non-programmers to read and enables quick and easy translation

to any computer language.

Writing pseudocode is more concise and easier to modify, so changes to the design can be

easily incorporated.

And lastly, unlike some flowcharts, pseudocode is usually less than one page.

In this video, you learned that:

Organizing and planning out software design enables programmers to write cleaner and more

reliable code, and organized code is very important from a readability, maintainability,

and scalability standpoint.

Two main methods of organizing and planning software code are by developing flowcharts

or writing pseudocode.

A flowchart is a pictorial representation of an algorithm showing the steps as boxes

of various kinds connected by arrows that indicate their order.

And the main goal of pseudocode is to explain exactly what each line of a program should

do, making the code construction phase easier for the programmer.

Welcome to Insiders’ Viewpoints: Types of Languages.

In this video, we will hear from experts discussing if they prefer developing in compiled or interpreted

programming languages.

most of the time, it doesn't really matter.

It just comes down to personal preference, like what are you going to get moving on the

fastest.

there are certain situations where it will make a difference, if you're compiling ahead

of time versus interpreting on the fly, if you're trying to eke out, like very small,

incremental performance benefits,

I prefer compiled code or compiled language that's compiled language just because I have

a more of a guarantee that it will well maybe not necessarily work, but will do what I

thought it will do.

And at least we'll deploy properly.

When when when being rolled out.

but usually no, it doesn't, doesn't make that much of a difference to either the development

process or the like deployment and use process.

There.

They, they definitely have advantages and disadvantages to each of them.

But most of the time, doesn't make that much of a difference to what I'm doing.

I think, interpreted languages can you know, there's a lot of like flexibility there.

And often you can do more creative things.

And then you know, the compiler, it's sort of stops, the buck stops with the compiler,

and you can't really avoid compiler errors with the compiled language with interpreted

there's more interesting runtime possibilities there.

So I think it really depends on what environment you're developing for, or what your your risk

factor is, when it comes to being okay with, with potentially buggy code being out in the

wild.

What do you think are the benefits of object-oriented programming over procedural programming?

Anything can be an object.

And object-oriented design puts hierarchies and patterns onto those things.

And that structure can make it easy to keep designs in mind.

And keep in mind how the different objects interact with each other.

It's also very easy in object-oriented programming to paint yourself into a wall where you have

put too much structure around everything.

And you now have this like explosion of useless boilerplate code.

So there's a bit of a, not catch 22.

But just a balance that you need to find in between how much structure should your thing

have?

How much flexibility should it have?

And how much should your intentions be imposed on that design and the way that people interact

Play video starting at :3:6 and follow transcript3:06

I think procedural programming can be, you know, feels more mathematical and more sort

of, like pure engineering, in some sense.

And I think that that also can lend itself well to someone who is really looking to not

just build software, but really wants to sort of feel like they're engineering the code,

in some sense.

It can feel like you're, you're kind of like, in the machine if you will.

So, I know that's a weird answer.

But it's, it's I think it doesn't, you know, depending on how your brain works to one can

lend itself more easily than the other.

So, but I love I think it's worth trying them both out.

if you get super object-oriented, you can get a little to a little too prescriptive

with the designs, I think.

So that's just whenever you're using objects, it's just a thing to keep in mind that you

can go, you can go too far with it.

But where that line of too far is kind of depends on exactly what it is that you're

building, sometimes there, sometimes the sky's the limit, just regulate everything.

And sometimes the amount of flexibility that you require actually does mean that you can't

really have an object of any sorts.

Most of those situations should be pretty weird, those would be on the extremes.

But most of the time, a few simple objects will do a lot of good, no matter what the

rest of your system looks like.

Yeah, I think object-oriented programming to me is, for me, personally, I think it does

lend itself well to like real-world mappings and data models.

And, you know, the sort of classic examples of you might, in your first object-oriented

software, demo, build something like a library that has books, and it has a checkout.

So, like you get there's physical objects that the way you learn it is very much sort

of you think about the software as something real worlds that helps you make the jump from

building something physical with your hands to building something with, with bits and

bytes.

Welcome to Branching and Looping Programming Logic.

After watching this video, you will be able to:

Describe the two types of programming logic

Define Boolean expressions and variables

and list the differences between branching and looping.

There are two major types of programming logic: branching and looping. Both types use Boolean

expressions and variables:

A Boolean expression is a type of programming statement with only two values, either "true"

or "false."

And variables have assigned values that are passed into a function or subroutine within

a more extensive program.

Computers use Boolean logic to make decisions. The computer takes one action if a Boolean

expression is true and a different action if the expression is false.

Typically, a program consists of instructions that tell the computer what to do and data that

the program uses when it is running. A variable has a value that can change, depending on

conditions or information passed to the program. Boolean logic, along with variables, form

the basis of programming.

Branching logic is where a computer program makes a decision following a different set

of instructions, depending on whether certain conditions are met during the program's execution.

Each possible code pathway creates another branch. The branch of code that runs depends

on the values assigned to the parameters of the branching procedure. There is no limit

to the number of branches to implement complex logic.

The values of these parameters may be input by the user or generated by the output from

a previous procedure.

Branching contains constructs that occur and are processed to determine the path a program

takes when running.

Branching statements (also known as constructs) allow the execution flow to jump to a different

part of the program. The common branching statements used within other control structures

include:

if,

if-then-else,

Switch,

and GoTo.

The if statement is a decision-making construct that guides a program to make decisions based

on specified criteria. The if statement executes one code set when a specific condition is

met (TRUE) or another code set if the condition is not met (FALSE).

The if-then-else is a conditional construct that executes its substatement, which follows

the “then” keyword. This only occurs if the provided condition is true.

The if-else statement extends the “if” statement by specifying an action if the

“if” (true/false expression) is false. With the if-else statement, the program will

execute either the true code block or the false code block, so something is always performed

with an if-else statement.

In computer programming languages, a switch statement is a type of selection control

mechanism used to allow the value of a variable or expression to change the control flow of

program execution via search and map.

GoTo is a statement found in many computer programming languages that performs a one-way

transfer of control to another line of code. In contrast, a function call typically returns

control.

Now, let’s look at the logic of looping programming. A loop is a sequence of instructions

that continually repeats until reaching a specific condition. Typically, a particular

process is performed, such as retrieving and changing data, and then some conditions are

checked, such as whether a counter has reached a prescribed number. If it has not, the next

instruction in the sequence is to return to the first instruction in the series and repeat

the sequence. If the condition is reached, the next instruction "falls through" to the

next sequential instruction or branches outside the loop. A loop is a fundamental programming

idea commonly used in writing programs.

There are three basic loop statements:

While

For

And Do-while.

In a While loop, a condition is evaluated before processing the body of the loop. If

a condition is true then and only then the body of a loop is executed.

In a For loop, the initial value is performed only once, then the condition tests and compares

the counter to a fixed value after each iteration, stopping the For loop when false is returned.

In a Do-while loop, the condition is always executed after the body of a loop. It is also

called an exit-controlled loop.

In this video, you learned that:

There are two major types of programming logic: branching and looping. Both types of logic

use Boolean expressions and variables.

Boolean expressions have only two possible values, either true or false, and variables

have assigned values that are passed into a function or subroutine within a more extensive

program. Variables have values that can change, depending on conditions or information passed

to the program.

And branching is deciding what actions to take, while looping is deciding how many times

to perform a certain action.

Welcome to Introduction to Programming Concepts – Part 1.

After watching this video, you will be able to:

Describe the purpose and use of identifiers in programming.

And Describe the purpose and use of containers in programming.

To fully understand software programming,

there are some fundamental programming concepts you need to know first.

The first concepts you will look at are identifiers.

Software developers use an identifier to reference a program component

such as a stored value, a method, an interface, or a class, by assigning a custom-named label to it.

If the identifier stores data, then the data values in the program can be one of two types:

either a constant, or a variable.

A constant is a data item whose value does not change within a program.

This could be a numerical constant such as the mathematical value of Pi,

or it could be a text string that remains constant such as a player’s name within a game.

Constants are also referred to as ‘named constants.’

You assign a value to a constant when you define it.

For example, you might want it to refer to a numerical value within your program such as Pi,

or a cost price, or a rate of tax, and then, rather than repeatedly retyping

the same numerical value, you can just declare a constant for that value instead,

and name it something meaningful like ‘pi\_value’, or ‘cost\_price’, or ‘tax\_rate’.

Play video starting at :1:42 and follow transcript1:42

There are a couple of major benefits to using constants in your programs;

one is for ease of readability in your code,

and the other is that if the specified value changes in the future,

you only need to change it once on the constant,

rather than finding every instance of that numerical value within your code and changing it.

The other type of identifier used in programming is a variable.

As the name suggests, this kind of identifier is not constant;

its value can change during the program’s execution,

such as a user entering their age in an application

or a high score in a game.

Variables can be strings of text, numerical values, or any other type of data.

Using a variable as an identifier is a useful way to refer to program items that are unknown to you,

such as a username, a service, or a file name for instance.

If you don’t use a variable, then you will need to hard code

all the names and values in your program, which is not considered best practice.

Variables can be declared and assigned a data type and initial value as they are defined,

or you can decide not to assign an initial value when you define a variable,

and instead have the value assigned later by instructions within the program.

In addition to the identifier data structures already discussed,

there are also special kinds of identifiers that can reference multiple elements in a program,

and these are referred to as containers.

Being able to specify multiple elements means that you don’t have

to create a variable for every individual element.

This makes it faster and more efficient.

For example, if you only want to store six numerical integers,

then you could argue that you could just create six variables, one for each.

But what if you need to store 1,000 integers or more? In such a case,

defining 1,000 variables would obviously be an unwieldly and inefficient programming technique,

so in this case you would use a container.

There are two types of containers to specify multiple elements: arrays and vectors.

The simplest of these container types is an array.

In an array, a fixed number of elements of the same type are stored

in sequential order, starting from index zero.

When you declare an array,

you specify the data type of the values it contains,

such as an integer, or boolean, or a string,

and then the maximum number of elements it can contain.

The syntax for declaring an array is to specify the data type first,

then the name of the array, then the maximum size of the array in square braces.

In contrast to arrays that have a fixed size, vectors have a dynamic size,

and they will automatically resize themselves as you add

elements to them or remove elements from them.

For this reason, you may also see them referred to as dynamic arrays.

Because they are dynamic in nature, vectors take up more memory space than arrays,

and their elements also take a little longer to access than elements in an array,

as they are not stored in sequential memory locations.

The syntax for declaring a vector is to specify the container type of the vector first,

then specify the data type in angle brackets,

then the name of the array. Note that because it is a vector, you do not need to specify

a maximum number of values it can contain because the size is not fixed, it is dynamic.

Play video starting at :5:26 and follow transcript5:26

In this video, you learned that:

Software developers use an identifier to reference a program component.

If an identifier stores data, then it can either be a constant or a variable.

A constant is a data item whose value does not change within the program.

A variable is not constant; it can change during the program’s execution.

In an array, a fixed number of elements of the same type are stored

in sequential order, starting from zero.

And vectors have a dynamic size,

and they automatically resize themselves as elements are added or removed.

Welcome to Introduction to Programming Concepts – Part Two.

After watching this video, you will be able to:

Describe the purpose and use of functions in programming,

and describe the purpose and use of objects in programming.

Next, let’s look at the fundamental concept of functions.

Functions are a consequence of the

modular programming software development methodology

that encourages the separation of a program into multiple modular components,

where each performs a specific task within a program.

So, a function is essentially a piece of structured, stand-alone, and reusable code

that will perform a single specific action.

This enables software developers to take a substantial,

complex program and divide it into smaller,

more manageable, and focused pieces.

Although some programming languages may refer

to them as something else,

such as subroutines, procedures, methods, or modules,

most modern programming languages refer to them as functions.

Functions take in data as an input, then process the data,

and then return the result as an output.

There are essentially two types of functions.

Standard library functions are the built-in functions provided by the programming language.

Common examples include the ‘If’, ‘Else’, ‘While ’ and ‘Print’ functions.

But programming languages also allow you to write your own functions.

And once you’ve written a function, you can use it over and over again.

The way that the blocks of code that make up a function are identified

is different across programming languages.

Some use braces, some use begin and end statements,

and others use indentations for example.

There are a few steps to using functions.

The first thing you need to do is define (or create) a function.

When you define a function, you provide a function keyword,

then give the function a unique name,

and you provide the statements that make up the body of the function.

Once a function has been defined, it then needs to be called (or invoked).

When you call a function,

the specified actions within the function are performed using any specified parameters.

While defining and calling functions are common to all programming languages,

some programming languages, such as C and C++ ,

also require you to declare a function.

Next, let’s look at the concept of objects.

Understanding what objects are is key to

understanding object-oriented programming.

Object-oriented programming (or OOP) is a

programming methodology that is focused on objects rather than functions,

which is what procedure-oriented programming is focused on.

The objects themselves will contain data in the form of properties (or attributes)

and code in the form of procedures (or methods).

The key distinction between the two methodologies is that

where procedural programming uses methods to operate on separate data structures,

OOP packages them both together, so an object operates on its own data structure.

Play video starting at :3:19 and follow transcript3:19

Consider the real-world objects in your life

such as your car, bike, TV, or your washing machine,

and ask yourself the following two questions:

“What states can the object be in?” and

“What behaviors can the object perform?”

When you think about your answers to these questions,

you will find that the answers to these questions vary across the objects.

More complex objects will typically have more potential states

they can be in and will be able to perform more behaviors.

Play video starting at :3:50 and follow transcript3:50

In programming, a software object is similar

to a real-world object, conceptually speaking,

in that they too consist of states (or properties) and behaviors (or methods).

Software objects can be anything, such as a Windows service, a user account,

a database table, or a system folder.

Objects store their properties in fields (referred to as variables in some programming languages),

and expose their behaviors through methods (referred to as functions in some programming languages).

Play video starting at :4:23 and follow transcript4:23

In this video, you learned that:

• A function is a piece of structured, stand-alone, and reusable code that

will perform a single specific action.

• The defining and calling of functions is common to all programming languages.

• Object-oriented programming is a programming methodology

that is focused on objects rather than functions.

• Software objects consist of properties and methods.

***Module 4***

Welcome to Introduction to Software Architecture. After watching this video, you will be able to:

Describe what software architecture is;

Explain the importance of well-designed software architecture;

Explain how software architecture impacts design decisions such as tech stacks and production

environments; and

List several artifacts from software architecture design.

Software design and documentation take place during the design phase of the SDLC.

Software architecture, simply put, is the organization of the system.

Software architecture serves as a blueprint for the software system that the programmers

use to develop the interacting components of the software.

The architecture comprises the fundamental structures of a software system and explains

the behavior of that system.

The architecture defines how components should interact with each other, the operating environment,

and the principles used to design the software.

The software architecture captures early design decisions that are often costly to change

once implemented.

A software’s architecture addresses non-functional aspects of the application such as performance,

scalability, maintainability, interoperability, security, and manageability.

Well-designed software architecture is important for a number of reasons.

First, it balances the differing needs of the stakeholders and serves as a basis for

communication among team members.

Next, the architecture represents the earliest design decisions, and those decisions conflate

other coding implementation decisions later in the development process.

Also, the well-designed architecture allows for agility due to changing requirements.

A well-organized architecture increases the lifespan of the software system even when

implementation details change.

Architectural design also guides the choice of technology stacks used for the system.

Remember that architecture addresses non-functional capabilities so choosing stacks that address

these requirements is paramount in the design phase.

Recall that a tech stack is a list of all the technologies including software, programming

languages, libraries, and frameworks that will be used to create the system.

The architects must be aware of the stack’s advantages and disadvantages to anticipate

development needs.

Much like blueprints communicate design decisions to the builders of a house, there are also

several artifacts produced during the architectural design phase that are used to communicate

the software design to the stakeholders.

These artifacts include a software design document, or SDD, an architectural diagram,

and unified modeling language, or UML, diagrams.

The SDD is a collection of technical specifications that indicate how the design should be implemented.

It provides a functional description of the software and design considerations such as

assumptions, dependencies, constraints, requirements, objectives, and methodologies.

The architectural diagram displays components, their interactions, their constraints, and

their confines.

It displays the architectural patterns used in the design.

Architectural patterns are general, reusable solutions to commonly occurring problems and

will be discussed in more detail in an upcoming video.

UML diagrams are diagrams that communicate structure and behavior using common programming

language agnostic notation.

UML diagrams will also be discussed in more detail in another video.

Another topic to be discussed in this module as it relates to software architecture includes

production deployment considerations.

The architecture drives choices about the environment in which the software is released.

The production environment is comprised of the infrastructure that runs and delivers

the application to the end-user such as the servers, load balancers, and databases.

In this video you learned that:

Software architecture functions as a blueprint and represents the underlying organization

of the application.

A good architectural design is important because it serves as a basis for communication among

team members.

Software architecture represents the earliest design decisions, is hard to change once development

starts, and accommodates changing requirements during development.

Architectural design influences technology stack choices and the production environment

and

Artifacts resulting from the design include the SDD, the architecture diagram, and UML

diagram.

Welcome to software design and modeling.

After watching this video, you will be able to:

Compare and contrast the terms “structured design” and “behavioral models.”

Describe Unified Modeling Language (or UML) and its advantages when designing software.

And, discuss the purpose of an interaction and a state transition diagram.

Software Design is a process during which

structural components and

behavioral attributes of the software are documented before it can be developed.

One of the key activities of the design process is modeling the software to express its design.

This involves creating visual or diagrammatic representations of the bigger software solution, and its sub-components,

as well as the interactions between them.

This can be done using simple flowcharts or more standardized methods like UML.

A software system can be construed in terms of structural elements. Structured design

conceptualizes a software problem into well-organized smaller solution elements called modules and

sub-modules.

Structured design stresses organization in order to achieve a solution. A well-structured

design should contain modules that are cohesive and loosely coupled.

Cohesion means that all functionally related elements are grouped together.

Coupling is the communication between different modules. For a system to be loosely coupled

the modules should be weakly associated so changes in one component have minimal effect

on another. Loose coupling is an architectural principle often used in service-oriented architectures

and microservices based architectural patterns, which will be discussed later in the module.

The diagram shows a simplified example billing system. Modules are arranged in a hierarchy

and communicate with each other.

The rectangles represent the modules and sub-modules.

You can see that “billing” is the main module, and the other rectangles are

sub-modules to the main billing module. In this example the sub-modules are “insurance

verification,” “submit claim,” and “output total.”

The arrows represent the flow of the data in the system.

Behavioral models describe what a system does, without explaining how it does it.

The overall behavior of a system can be communicated through behavior models.

There are a number of different UML diagrams that can be used to communicate the behavior

of a system. We will discuss two such diagrams,

a state transition diagram, and

an interaction diagram.

When developing a complex software system with interconnected modules, it can be difficult

to remember the relationships, behaviors, and hierarchies among different elements.

UML, which stands for Unified Modeling Language, is a way to visually represent the architecture,

design, and implementation of complex software systems. UML is a standardized modeling language

that can be used throughout development processes.

UML diagrams can be divided into two classes: either structural or behavioral.

UML is programming language agnostic, so software developers can readily interpret and apply

it to their work no matter which language they are developing in.

There are several advantages of using UML to communicate architecture, behavior, and

structure with development teams. The biggest advantage of UML diagrams is that they allow

you to

plan out features before any coding takes place which saves

time and money. Secondly, the diagrams can be used to

bring new team members or developers switching teams to get up to speed quickly.

Also, the diagrams can be used to facilitate communication between technical and non-technical

audiences more easily.

And finally, having a visual representation of the system allows developers to navigate the

source code because they can see the relationships among modules. So there are many types of UML

diagrams but generally, UML diagrams can be classified as either behavioral or structural.

We will discuss behavioral models next and then object oriented-design.

The behavior of a system can be explained and represented with the help of a UML diagram

called a state transition diagram. The state transition diagram is a collection of states

and events that describes the different states that a system has and the events which cause

a change of state in the system. The diagram shown is an example of a state transition

diagram that models a patient going to see a doctor at a clinic. The different states

include

“waiting,”

“testing,” and

“with the doctor.” The arrows represent possible transitions from one state to another

and names the event that triggers the transition.

An interaction diagram is used to model the dynamic nature of a software system. They

help visualize objects and their relationships. A sequence diagram, which is the type of interaction

diagram shown here, displays the communication between objects with respect to time. This

example shows a patient making an appointment in an online portal. This is another example

of a behavioral UML diagram.

In this video you learned that:

Structured design breaks down a software problem into well-organized smaller solution elements.

Behavioral models describe the behavior of the system without explaining how the system

implements the behavior.

Developing UML diagrams saves time and money by helping developers quickly get up to speed

on a project, plan features in advance of coding, and help developers navigate source

code easily.

A state transition diagram is a behavioral model containing a collection of states and

events that describe the different states of a system and the events which cause a change

of state.

And finally, an interaction diagram describes how interacting objects communicate.

Welcome to object-oriented analysis and design.

After watching this video, you will be able to:

Explain what objects and classes are.

Describe the purpose of a class diagram, and

explain object-oriented design in relation to software architecture.

Object-oriented analysis and design, or OOAD for short, is an approach for analyzing and

designing a software system when the system will use

object-oriented programming languages to develop it.

So, before we discuss OOAD, let’s learn a little more about object-oriented programming

in languages like

Java,

C++, or

Python.

At the heart of OOAD are objects.

Objects contain data, and they also have behaviors that prescribe the actions the object can

take.

I could create an object for example, that represents a patient.

Let’s say the patient’s name is Naya Patel and Naya needs to cancel an appointment she

made.

Before creating Naya, however, we must first create a generic version of a patient object.

The generic version of an object is called a “class”.

Let’s discuss classes next before we discuss more about Naya.

Specific objects, also called instances, are created from “classes”

which are blueprints or templates for an object.

So, from our previous example, considering Naya Patel, Naya would be an instance of the

patient class.

The class contains the object’s generic attributes – the properties and methods

– but it is only when the object is created, which is called “instantiation,” inside

the code that these generic attributes are set to particular values.

So, the patient class might have a variable called LastName, which is a property but does

not specify what that last name is.

LastName is just a placeholder until the object is created and assigned a name.

And once the object has been instantiated its methods can be called to make the object perform

some action such as making or canceling an appointment.

OOAD is used for a system that can be broken down into objects that interact with each

other.

In this way, multiple developers can work on different aspects of the application at

the same time.

As noted earlier, visual UML diagrams can be created that show both the static structure

and dynamic behavior of a system.

We’ve seen a couple of examples of behavioral UML diagrams already, so now let’s look

at a structural UML diagram called a “class diagram” next now that we know what classes

are.

This is an example of a “class diagram.”

Class diagrams are commonly used to communicate a software system’s structure in OOAD.

The class diagram shows how the classes in an object-oriented design relate to one another.

Each box represents a class and shows its attributes.

Recall that an object’s attributes are both its properties or its data, and its available

actions, called methods.

A class diagram also shows the relationships between classes.

A subclass is said to “inherit” its parent class attributes meaning it has the same properties

and methods as the parent class but also may add additional properties and methods.

In this diagram,

the nurse, doctor, and technician classes are subclasses of

medical personnel, and the

specialist class is a subclass of the doctor class.

This means that doctors can do anything medical personnel can do and specialists can do anything

a doctor can do.

In this video you learned that:

Object-oriented analysis and design is the process of planning a software system based

on the behaviors of interacting objects.

Objects contain data, and they also have behaviors that prescribe the actions the object can

take.

Classes are blueprints for objects.

And a class diagram is a structural UML diagram that shows the relationship between objects.

Welcome to Insiders' Viewpoints: The Importance of Design and Software Architecture.

In this video, we will hear from experts discussing the importance of design and software architecture

in a software engineering project.

Design and architecture are crucial.

If you don't know where you're going, how do you know what you need to do now?

You always need to know what you should, what your program needs to be prepared to handle,

and the system that it's operating within.

If you are running a series, if you're running a bunch of applications that are all co-hosted

on the same server, you can make certain assumptions about the ways those services can interact

with each other that do not hold.

I also sometimes think of architecture and orchestration.

So you can you can have like a beautiful, really nice sounding instrument.

But if it's not in sync with the rest of the orchestra, it's going to sound like a complete

cacophonous mess in the end.

So you really need to think about the bigger picture.

Sustainability, whether you're, you know, what you're building actually makes sense

in the larger sort of ecosystem.

And every time if you spend the time to write your code, and then you figure out why it

doesn't work, you've wasted a lot of time writing that code, maybe you've learned some

things, it's not a total waste.

But you've spent a lot of time that could have been spent moving forward.

And so if you can think about those things ahead of time on a project.

And then when you write your code, you're reasonably sure that the code you write is

actually going to persist and get used, that's generally speaking a better place to be in.

So architecture is very important.

like there's a lot of talk about its scale, global availability, often something you build

would work very well in a very hyper local environment, but not work well, when it's

expected to be used across the globe or, you know, works well for one user, but doesn't

work well, for a million users.

I'm building a service, I need XYZ pieces of data to come into the function that

I want to write.

Where are those things coming from?

How are they getting passed in?

Once I have them what do I need to do with them?

What else needs to access this data?

And how am I going to get the data to those other parts also?

If you don't have those questions answered, you end up writing a lot of churny code.

If everything is on a separate server somewhere in the cloud, or separate physical servers

that are just located somewhere different inside your data center, then you have to

incur a network hop to go over them.

And that takes time, your page might start timing out and your customers might get really

upset at you for how long things take.

And so, if you're building something in like a microservice architecture, the way that

you load your data becomes much more important than if everything is hosted together on the

same server.

But maybe the question of who is using my service?

How do I know who you are?

That can become an interesting question, depending on where your servers where your services

are running, and what mechanisms are available to you to identify people.

If your people if you want your customers to log into a website, how are you doing account

management?

How do you know who that person is and what they're allowed to do?

Or if you have multiple services talking to each other again, how do you know what that

service is?

What it's allowed to do?

How do you make the call between one and the other?

How do you transmit data between different parts of your stack?

So architecture is very important. It doesn't always have to be super well defined, super

strictly defined, but you need to have some idea of what what things do you need, roughly

where they're going to live.

And what things what are they responsible for?

Those would probably be the three big questions to ask, what, where and scope of them?

So these sorts of concepts, you know, you have to think about those things.

When you're designing or architecting a system, you're thinking like, 5, 10 years ahead to

when you're thinking about architecture and design, you're not thinking about just building

something quickly, in getting it up, you actually want to plan ahead.

So I think a lot of architecture and design is thinking about, you know, how the how many

steps, how many years, whatever in the future, you know, what is it going to look like then?

What do you need to do to keep growing the system and evolving it?

Because you know, you don't want to make gigantic architecture changes that often, you know,

you don't really use when buildings are repaired, they there's like, tiny bits of things get

repaired.

And then at some point, you tear the thing down and rebuild it.

And I think that's a similar model applied to sustainable system architecture, you don't

want to re-architect and redesign every few years, you want it to last as long as you

possibly can.

Welcome to Approaches to Application Architecture.

After watching this video, you will be able to:

Describe component-based architectures and service-oriented architecture.

Explain the characteristics of a component.

Differentiate between components and services.

And, describe the characteristics of distributed systems.

Play video starting at ::27 and follow transcript0:27

A component is an

individual unit of encapsulated functionality that

serves as a part of an application in conjunction with other components.

There are six characteristics of components. Components should be

reusable,

replaceable,

independent,

extensible,

encapsulated, and

non-context specific. Let’s briefly discuss each of these principles.

Reusable implies that components should

be designed such that they can be reused in different applications.

Replaceable means that a component should be easily replaced with another component.

Independent means the component should be designed so it doesn’t

have dependencies on other components.

Extensibility entails the ability to add behavior to a component without changing other components.

Encapsulation consists of bundling a component’s data and methods to hide its internal state,

so it doesn’t expose its specific implementation.

Creating a component that is non-context-specific involves designing it so it operates in different

environments. Data that sets its internal state should be passed

to the component rather than included within or accessed by the component.

Let’s take a look at some examples of components.

An API can be packaged as a component, if it can be reused across multiple systems and applications.

For instance, a component could be an open-source

API that connects a system to a particular database.

A component can also be the interface for a database, called a data access object, that

switches the user to a different database without the application knowing about the switch.

And a controller is a type of component that determines which other components need to

be called for a particular event. It controls the flow of data between two other components.

Play video starting at :2:25 and follow transcript2:25

Component-based architecture, then,

focuses on the decomposition of the design into these logical components.

Component-based architecture provides a higher level of abstraction than object-oriented designs.

A component-based architecture should define, compose, and implement loosely

coupled independent components so they work together to create an application.

A service is like a component, also a unit of functionality, but it is designed to be

deployed independently and reused by multiple systems.

A service focuses on a solution to a business need.

A key difference between a component and a service is that a service will only have one unique,

always running instance with whom multiple clients communicate. This diagram displays

the relationship between objects, components, and services in a layered architecture.

Services are made of components and components consist of objects.

A service is a type of component. It is meant to

be deployed independently of the overall system.

Examples of services include:

checking a customer’s credit,

calculating a monthly loan payment,

or processing a mortgage application.

In a service-oriented architecture, or SOA,

services are loosely coupled and interface with each other

via a communication protocol over a network.

SOA supports building distributed systems that deliver services to other applications

through the communication protocol. So, let’s discuss distributed systems next.

A distributed system is a system with

multiple services located on different machines that

coordinate interactions by passing messages to each other

via a communication protocol such as hypertext transfer protocol,

also known as HTTP. Even though the services on a distributed system operate on multiple machines,

a distributed system appears to the end-user as a single coherent system.

Play video starting at :4:31 and follow transcript4:31

A distributed system shares resources such as hardware, software, and data.

They are fault-tolerant, meaning if a node or a service fails the system continues to run

also implying that the system may change during execution without service interruption.

Multiple activities run concurrently on a distributed system

reducing latency and increasing throughput.

Another property of distributed systems is that they are scalable as the number

of users increases. The computers running the distributed system do

not need to use the same kind of hardware or operating systems.

A distributed system may be made up of different kinds of computers and programming languages.

Play video starting at :5:17 and follow transcript5:17

Now a node is any device on a network that can recognize,

process, and transmit data to other nodes on the network.

A distributed system consists of multiple

interconnected nodes where the nodes are running one or more services in an SOA.

Distributed systems generally use one or more of the following basic types of architecture:

client-server,

three-tier,

peer-to-peer or

microservices. These application architectures

and their characteristics will be discussed in another video.

Play video starting at :5:53 and follow transcript5:53

In this video, you learned that:

Components are reusable, independent, replaceable, extensible, encapsulated, and non-context specific.

Component-based architecture is the decomposition of the system into logical independent components.

Services are made of components and components consist of objects.

Services are deployed independently and can be reused by multiple systems.

In an SOA services are loosely coupled and interface with each other

via a communication protocol over a network.

And lastly, distributed systems run on multiple services on different machines and

they appear to the end-user as a single coherent system .

Welcome to architectural patterns in software!

After watching this video, you will be able to:

Describe different software architectures such as 2-tier, 3-tier, peer-to-peer, event-driven,

and microservices.

And provide an example of each architectural pattern.

An architectural pattern is a repeatable solution to a problem in software architecture.

Patterns highlight common internal elements and structures of a software system.

Different architecture patterns may share related characteristics.

In this video, we will discuss

2-tier,

3-tier,

peer-to-peer,

event-driven, and

microservices.

However, there are many other patterns that will not be covered in this video.

These patterns include

model-view-controller,

message-broker,

blackboard,

pipe-filter, and

controller-responder.

The 2-tier architecture, also called client-server, is a computing model in which the

server hosts, delivers, and manages most of the resources and services

delivered to the client.

The interface resides on the client machine and makes requests to a server for data or

services.

This type of architecture usually has

more than one client computer connected to a server component over a network connection.

A 3-tier architecture, or an n-tier architecture where there are more than three layers, is

the most common software architecture.

The 3-tier architecture is composed of several horizontal tiers that function together as

a single unit of software.

A tier only communicates with other tiers located directly above and below it.

Related components are placed within the same tier.

Changes in one tier do not affect the other tier.

The 3-tier architecture organizes applications into three logical and physical computing

tiers:

the presentation tier, or user interface

the middle tier which is usually the application tier, is where business logic is processed

the data tier, where the data is stored and managed

The peer-to-peer architecture, or P2P for short, consists of a decentralized network

of

nodes that are both clients and servers.

The workload is partitioned among these nodes.

Peers make a portion of their resources directly available to other network participants, without

the need for central coordination by servers.

Resources are things like processing power, disk storage, or network bandwidth.

Peers both supply and consume resources, in contrast to the traditional client-server

architecture in which the consumption happens strictly by the client and the servers, supply

the resources.

Peer-to-peer architecture is useful for file sharing, instant messaging, collaboration,

and high-performance computing.

An event is anything that results in a change of state.

An event can be thought of as an action that is triggered by the end-user, such as a mouse

click, or another part of the program.

Event-driven architecture focuses on

producers and

consumers of events.

Producers listen for and react to triggers while consumers process an event.

The producer publishes the event to an event router.

The router determines which consumer to push the event to.

The triggering event generates a message, called an event notification, to the consumer

which is listening for the event.

The components in event-driven architectures are loosely coupled making the pattern appropriate

for use with modern, distributed systems.

Microservices are an approach to building an application that breaks its functionality

into

modular components called services.

An application programming interface, also called an API, is the part of an application

that communicates with other applications.

An API defines how two applications share and modify each other’s data.

APIs can be used to create a microservices-based architecture.

The API Gateway routes the API from the client to a service.

Orchestration handles communication between services.

Let’s discuss an example for each of these patterns.

A text messaging app is an example of a 2-tier pattern.

The client initiates a request to send a text message through a server and the server responds

by sending that message to another different client.

Another example of the 2-tier pattern, is Database clients connecting with database

servers.

Many web apps use the 3-tier pattern.

They use a web server to provide the user interface, an application server to process

user inputs, and a database server that handles data management.

The functions of these three types of servers will be discussed in more detail in another

video.

Ride-sharing apps such as Lyft and Uber are examples of event-driven patterns.

The customer sends a notification that they need a ride from a particular location to

another location, and that event is routed to a consumer.

Cryptocurrencies such as Bitcoin and Ethereum use a peer-to-peer pattern.

Each computer in the blockchain acts as both server and client.

Finally, social media sites are composed of microservices.

A user has an account.

That account can request different services such as adding friends, targeted ad recommendations,

and displaying content.

Architectural patterns are not necessarily mutually exclusive.

In other words, two or more of these patterns can be combined.

For instance, a three-tiered architecture can also be microservice-based, or a

peer-to-peer architecture can also be event-driven.

However, not all architectural patterns can be used in conjunction with others.

A peer-to-peer cannot also be two-tier because a single machine in a peer-to-peer architecture

represents both a client and a server whereas a two-tier architecture separates the client

from the server.

It is up to the system architect to determine which architectural patterns the software

system should adhere to.

In this video, you learned that:

An architectural pattern is a repeatable solution to an architectural problem

A 2-tier pattern has a client and server.

Text messaging apps use a 2-tier pattern.

A 3-tier pattern has 3-tiers that interact with each other.

Web apps use a 3-tier pattern.

An event-driven pattern has actions that are produced and responded to by a consumer.

Ride-sharing apps use an event-driven pattern.

The peer-to-peer pattern consists of a decentralized network of nodes that act as clients and servers.

Cryptocurrency is an example of the peer-to-peer pattern.

Microservices are loosely coupled individual services that behave as a single system and

interact with the client.

Communication is orchestrated among services.

Social media sites are an example and

Two or more patterns can be combined in a single system but some are not mutually exclusive

Welcome to Application Deployment Environments. After watching this video

you will also be able to:

List different types of pre-production environments and state their purpose.

Differentiate a production environment from other application environments.

And compare and contrast deployment options.

An application environment is the combination of the hardware and software resources required

to run an application.

This includes: the application code and/or binary executables

for its various components or modules

the software stack it requires for running the application such as modules and libraries

it depends on, third party applications and middleware, and the operating system,

any networking components and infrastructure, as well as

any physical or virtual hardware including computing or processing resources, memory,

and storage.

There are a variety of environment types depending on the application’s stage in the lifecycle.

The pre-production environments are those platforms that the application resides on

in various forms as it gets prepared for production. Common pre-production environments are

“development,” “QA,” which stands for quality assurance, and

“staging.”

The development environment is the platform on which the application is being actively

coded, and in many cases it may just be the developer's workstation.

The QA environment, sometimes called “testing” is the environment that allows the QA team

to test the application’s components.

The staging environment is the environment that is as close to replicating the production

environment as possible but is not meant for general users. Now, let’s discuss the production

environment.

The production environment, often just called “production” includes the

entire solution stack consisting of both hardware and software on which the application runs

as additional infrastructure components.

The production environment is intended for all users.

Unlike the pre-production environments, this robust environment must take the application

“load” into consideration because it is the environment intended for general use,

possibly by thousands or millions of people at the same time for enterprise-level applications.

Production environments must also take into account non-functional requirements like security,

reliability, and scalability.

This makes the production environment more complicated than the pre-production ones.

Now, let’s take a look at the production environment infrastructure.

There are several options for deploying application environments.

In on-premises deployment, the system and its infrastructure reside in-house, within

the organization’s physical location, often behind a firewall. Firewalls prevent unauthorized

access to or from a private network.

If an organization desires greater security or control of an application and the data

in use by that application, it may deploy the application on-premises.

For on-premises software deployments, an organization is responsible for the system, hardware, related

infrastructure, and maintenance required to run the application.

On-premises deployment is usually more expensive when compared to cloud deployment. Let’s

discuss cloud deployments next.

There are three types of Cloud deployment models—public, private, and hybrid.

The public cloud is when you leverage the software’s supporting infrastructure over

the open internet on hardware owned by the cloud provider. That hardware and the associated

services are shared with other companies. Public cloud providers include Amazon Web

Services (AWS), Microsoft Azure, Google Cloud Platform, and IBM Cloud. The public cloud

is the most common due to its scalability and cost.

Play video starting at :4:17 and follow transcript4:17

With a private cloud, the cloud infrastructure is provisioned for exclusive use by a single

organization. The software system can be run on-premises, or the infrastructure could be

owned, managed, and operated by a service provider. For example, AWS is also a private

cloud service provider. The main advantage of a private cloud is increased security,

but it also allows for more flexibility because it can be fully customized.

Play video starting at :4:47 and follow transcript4:47

A mix of both public and private clouds, working together seamlessly, is called a hybrid cloud

model. A hybrid cloud potentially optimizes the advantages of both public and private

cloud models with regard to cost, security, scalability, and flexibility.

Play video starting at :5:7 and follow transcript5:07

In this video you learned that:

Application environments include: development, testing or QA, staging, and production.

Production environments must also take into account non-functional requirements like load,

security, reliability, and scalability.

And application environments can be deployed either on-premises on traditional hardware, or on

public, private, or hybrid cloud platforms.

Welcome to Insiders’ Viewpoints: Deployment Architecture.

In this video, we will hear from experts discussing

important aspects of software deployment.

Play video starting at ::17 and follow transcript0:17

I think often you need to think about scale, like this sort of just

in general, is your system, how many users your system has?

How much data do you expect to flow through it?

How much data do you expect to produce and store?

How real-time do you need the data access to be?

What is your logging story like? Do you have privacy concerns?

Do you need to ensure that data is anonymized?

So you kind of, I think, thinking about the scale, the regionality, is it global?

Is it local?

Play video starting at ::51 and follow transcript0:51

What is your logging?

What are your logging needs, both for internal and external

data analysis, mapping out the data flows?

So these are sort of the things I think we start with is with design principles.

Play video starting at :1:3 and follow transcript1:03

And in terms of choosing infrastructure, like whether you want to go serverless,

whether you want to use a hosted back end, or you're hosting your own,

you know, these are all design and architectural decisions, that

will probably, for your first few jobs, already be taken for you.

What's your system

SLOs, your availability look like? Service level objectives,

right? So, you need to think about those sorts of things, I think, ensuring you,

Play video starting at :1:33 and follow transcript1:33

you have you like, what does a healthy system mean to you?

What does a well-running system mean to you, you need to define those metrics upfront?

And sort of have those monitored, and, you know, watch those as, as your system lives on and,

and make sure that you're meeting those objectives and goals and have a plan to mitigate them

if they fall below where you'd like them to be?

These days, I think microservices, and keeping code as modular-alized as possible to avoid,

you know, giant, sort of bloated behemoth code bases

is really important so that people can move fast and deploy fast.

But what I would say is a critical part of the deployment option is,

you know, when you look at deployment and integrating your code, testing is fundamental.

So I would strongly urge all of you to really, really pay attention to testing

and to use test driven development, TDD, test driven development, as a development habit that

you inculcate in your life as developers and engineers, it'll stand you in very good stead.

I think that's the single most important part,

you won't always do it, but you should try and do it as much as you can.

You need to think about whether you want to canarying the,

there's a canary in a coal mine analogy.

So there's a process called canarying where you

try to roll out new code and test whether it's going to blow things up or break things and

you want to think about, you know, you need to think about how quickly you need to release or not.

How quickly you may need to roll back your code or not.

***Module 5***

Welcome to What Does a Software Engineer Do?

After watching this video, you will be able to

Describe the types of software that software engineers develop and the technologies that

they use

Identify the categories of software engineers and the teams they work in, and

Describe the day-to-day tasks and responsibilities of a software engineer

Software engineers use their talents in engineering, mathematics, and computing to design and develop

software that solves real-world problems for their users.

If you’re an analytical thinker who enjoys problem solving, software engineering might

be the career for you!

Software engineers develop a plethora of different types of software, from desktop to web applications,

to mobile apps, games, operating systems, and network controllers.

They use many technologies to do this, including programming languages, development environments,

frameworks and libraries, databases, and servers.

There are two categories of software engineer:

Back-end engineers, also known as system developers, who build the computer systems

and networks that front-end applications use.

And front-end engineers, or application developers, who are more client focused.

They create the software that users will interact with, such as Android, iOS, and Windows applications,

and platform agnostic websites.

Software engineers work in a wide variety of settings.

You could work in a team developing:

Off-the-shelf software,

Bespoke software to meet a specific client’s requirements, or

Internal software for users in your organization

And within your team, you might work on:

Data integration layers which access and load data from a variety of sources into your solution,

Business logic which applies real-world business rules to the data in your solution, or

User interfaces which enable users to interact with your solution.

Play video starting at :2:11 and follow transcript2:11

On a day-to-day basis you might undertake tasks such as:

Taking user specifications and designing new software systems to meet their requirements

Writing code and testing that it works as expected

Evaluating and testing new software programs

Optimizing software programs for maximum efficiency

Maintaining and updating existing software systems

Documenting code so that other developers can understand it or,

Presenting new systems to users and customers

And some software engineers, such as DevOps practitioners, also integrate and deploy their

code onto the underlying infrastructure. As well as building your own systems

you'll also be testing, improving, and maintaining software built by your colleagues.

The responsibilities of a software engineer are as many and varied as the tasks which

they undertake.

In a junior position you are likely to start out with a limited set of responsibilities

focused on writing, testing, deploying and documenting code, but as your career progresses

this will widen.

And, in a senior role, you will likely have primary responsibility for multiple areas

of the software solution including the planning and designing phases.

In this video you learned that:

Software engineers design and develop a range of software solutions

Back-end engineers build the computer systems and networks

And front-end engineers build the user interfaces

Software engineers undertake a range of tasks, from designing and writing new software to

maintaining and updating existing software

And as your career develops, you will move from being responsible for one small section of

code to many areas of one or more products

Hi, I’m Shelly and I work as a Software Engineer for a start-up company in North Carolina.

When I arrive at work in the morning, I spend some time catching up on my messages

and checking my calendar for the day.

Before heading to my daily standup meeting,

I quickly check the status of my code merge request from yesterday.

Great, it’s got positive feedback from others in the team and some suggestions

from my mentor about improvements I could make. I’ll work on those later.

Play video starting at ::40 and follow transcript0:40

During our daily standup meeting, each member of the team

talks about what they achieved yesterday and what they’re working on today.

Play video starting at ::48 and follow transcript0:48

Jonathan, my mentor, gives me some more advice

about how I can optimize the code I wrote yesterday.

Play video starting at ::56 and follow transcript0:56

After the meeting, I return to my desk excited to implement

the feedback I’ve received and continue working on my code.

Play video starting at :1:4 and follow transcript1:04

Now it’s focus time, when I can concentrate purely on my project work.

Play video starting at :1:10 and follow transcript1:10

I use Jonathan’s comments on the merge request alongside the notes that I made

during our standup to re-engineer a couple of methods to run faster.

Play video starting at :1:19 and follow transcript1:19

I also use some of our analysis software to compare timings with the previous version

and, when I’m satisfied that there’s a marked improvement,

I submit a new merge request to gather additional feedback.

Play video starting at :1:32 and follow transcript1:32

I then send Jonathan a message to ask if he can review it later today.

After lunch, I’m invited to a meeting where the marketing department

explains that they need a new feature to monitor the effectiveness of their campaigns.

Play video starting at :1:45 and follow transcript1:45

One of the senior software engineers, Marie, mentions that she’s previously developed a

React-based front-end for a management dashboard that we could easily reuse.

Play video starting at :1:59 and follow transcript1:59

I’m tasked with creating a minimum viable product

(MVP) for a small team in the marketing department to trial and provide feedback on.

Play video starting at :2:12 and follow transcript2:12

I’ve not used React before, but this is a great chance to get some hands-on experience with it

while still making a contribution to the team.

Play video starting at :2:21 and follow transcript2:21

Learning on the job and developing new skills is an important part of a Software Engineer’s role.

Play video starting at :2:29 and follow transcript2:29

On my return from the meeting, I find that Jonathan has reviewed my code and approved

the merge request. He’s impressed with the speed of the code and with my coding style.

Play video starting at :2:39 and follow transcript2:39

It’s really helpful to get regular feedback from more experienced members of the team

who can suggest alternative ways of solving problems.

Play video starting at :2:48 and follow transcript2:48

I’ve also received a bug report notification for one of the features I was working on last week.

When I review the full report, I quickly spot the mistake and begin fixing it.

Play video starting at :2:59 and follow transcript2:59

To check that the fix and the original functionality work as expected,

I develop and run a test case for the feature.

It passes, so I submit a merge request for my fix and resolve the bug.

Play video starting at :3:13 and follow transcript3:13

Now I’m free to start researching React. I spend the rest of the afternoon reviewing

the React documentation, watching videos, and reading blogs to give me an overview of React.

I then contact Marie to ask for access to the front-end she developed so I can review the

existing solution. This gives me design ideas and inspires me to start planning my dashboard.

Play video starting at :3:37 and follow transcript3:37

On my cycle home, my head is buzzing with thoughts about how the new dashboard

will look and the functionality I can provide through it. Tomorrow can’t come quickly enough!

Welcome to Skills Required for Software Engineering. After watching this video, you will be able to

differentiate between hard and soft skills,

describe the hard skills needed by a software engineer,

and describe the soft skills needed by a software engineer.

Hard skills are the practical skills needed to perform a particular role, so for a software

engineer these will be the technical skills they need to design, build, maintain, and

repair software solutions.

Hard skills are learned skills.

In the case of software engineering, they are usually learned in a school, college,

or university environment, or by studying online courses, diplomas, or certificates.

Alternatively, they can be gained from years of experience in the field.

Hard skills are quantifiable, so it’s easy to measure whether an individual can demonstrate

a particular skill and to certify them in that skill.

Commonly required hard skills in the software engineering sphere include programming languages,

version control, cloud computing, testing and debugging, monitoring, troubleshooting,

Agile development, and database architecture.

Now, Soft skills on the other hand are less tangible.

They are your personal characteristics and interpersonal skills.

They’re the non-technical skills that are part of your personality and as such, are

harder to define, quantify, or certify than hard skills.

Because they’re not linked to a particular business though, they are easily transferable

between roles and across industries.

The job requirements for any role will be a combination of hard skills and soft skills.

So, let’s look at some of the hard skills that are relevant to a software engineer.

Software analysis and design skills are essential for a software engineer.

You need to be able to analyze your users’ needs through a variety of methods and then

design effective solutions that meet those needs.

You also need to be able to develop those solutions.

Computer programming and coding are essential development skills.

While some job roles require a specific language and/or toolset, employers will sometimes welcome

you with experience in any language but expect you to cross-train into their preferred language.

The coding bootcamp website, Coding Dojo, states that some of the most in-demand languages

are currently Java, Python, C#, and Ruby.

An understanding of a variety of frameworks and object-oriented principles are also key

skills for a software engineer.

While you’d hope to always create flawless solutions, it is likely that at times your

code will either not work or not work in the way that you intended.

You need testing skills to determine whether your code meets the functional specification

of the solution and if it’s easy to use.

And when your code isn’t working as expected, you need debugging skills to work out why.

And when your solution is complete, you need deployment skills to distribute it to your users.

These could include

shell scripting,

containers,

and continuous integration and continuous delivery (or CI/CD) tools.

You’ll also need monitoring skills so that you can review the performance of your solutions

and troubleshooting skills to resolve any issues that may occur.

So now let’s look at the soft skills of a software engineer.

Teamwork is a key soft skill.

Software engineers work in a variety of teams, some based on the project they’re working

on and some based on their specific role.

If you’re practicing Agile development, you may also work in small teams known as

squads.

And you might also work closely alongside another developer in pair programming.

Working in teams enables you to take advantage of each individual’s strengths as well as

providing opportunities for you to gain new knowledge and skills.

You’ll need to be able to communicate with a wide variety of stakeholders in your project

– from technical colleagues to non-technical personnel.

For example, you may need to ask:

your peers for support and ideas,

your manager for guidance and direction,

your client for clarification of their needs,

and your users how they’d best interact with your solution.

Software solutions are often time-sensitive projects and as such, your manager will be

keen that you meet their deadlines.

Managing your own time is imperative to ensure that you don’t cause delays to others waiting

for your work.

And with increasing numbers of teams working across time-zones, what could previously only

seem like a small local delay can create a whole day of lost time for someone located

elsewhere in the world.

Software engineers need great problem-solving skills to succeed.

You need them:

in the design phase to work out how to create an appropriate software solution,

in the development phase to work out the code required to perform the task required,

in the testing and debugging phase to locate and resolve any bugs,

and throughout the lifecycle of the software to manage any

issues that may arise.

And when those issues do arise, you need to be adaptable to meet the changing needs of

the project.

Examples of changes include:

Your client requesting a change or addition to the functionality,

Your manager requesting you to move onto a different area to meet a looming deadline,

Or your user requesting an alternative method of achieving a specific task.

And finally, you need to be open to feedback on your work.

Most software teams use some type of peer review system where peers review each other’s

code.

This helps enforce any corporate standards and improves the code.

In a junior role, you’ll also likely have a mentor who’ll provide feedback and

pointers where you can improve.

And your stakeholders will provide feedback on pre-release and final versions of your

solution.

Accepting all of this feedback in an open and welcoming way ensures that your solution

becomes the best that it can be and that you progress in your role.

In this video, you learned that:

a combination of hard and soft skills are essential to a software engineer.

Hard skills are measurable, learned skills such as:

Programming,

testing,

and troubleshooting,

And, soft skills are your personality and characteristics, such as

communication

and problem solving.

Welcome to Insiders’ Viewpoints: Advice to Future Software Engineers.

In this video, we will hear from software engineers discussing their advice to those

who are considering getting into the field.

Practice, you should practice all the time, you should learn new skills as much as you

can.

There are a lot of free resources out there that you can use to help you build new projects,

I think the best way to continue working on your craft is to learn not only learn new

things, but also use them to build new projects.

It can doesn't matter if it's something that's been done before the point is that you're

doing it now.

And you're using the tools that you've used to build something new.

The most important thing for my career has been finding a business domain that I am interested

in, that has what that have looked like, not low hanging fruits, but interesting problems

to grapple with.

The interesting problems for you are certainly going to be the different than the interesting

problems for me.

But if you narrow down those, if you narrow down the domains that you're interested in,

that can really help tailor both of the things that you study the types of jobs that you

apply for.

I have other friends where their domain, the domains that they're super interested in is

like, like, machine learning, like they're just nuts for machine learning.

And so they'll do anything that they can exercise, like, an artificial intelligence kind of thrust

to the whatever business problem it is that they're working on.

So like find that general domain, and then kind of like, dive in on that one.

But don't be don't be scared to fail either.

I think the beauty about what's one of the one of the beauties about software engineering

and, and the way that tech is evolving, is there's there are a lot of opportunities and

it will continue to be new opportunities.

And you know, there's lots of movement in the industry.

So if you take a job and you really are excited about it, but you know, it ends up not working

out as well.

You can always think, you know, move to somewhere else.

There's lots of different types of lots of roles, lots of companies, all that sort of

people move around so don't don't don't let a one potential failure either mean, you give

up, I think it's wouldn't be a failure.

Yes, so my best advice for those looking to get into the software engineering industry

is to figure out what role you want to get into, make a list of the required skills for

that role and find free resources to start learning online.

So when I was exposed to coding in 2018, after trying a few technologies, I decided that

I wanted to be a front end engineer or a front end developer.

And so I made a list of all the skills that I needed.

And I started learning those skills using like free or low cost resources online.

I think being careful about what sort of hot and new there's, there's often this is more

specifically to like taking I think taking jobs and and how do you find like a good role

and fit for you?

I know a lot of companies will often sort of market themselves as being you know, we're,

we're we're working with XYZ new technology.

It's, it's super, super hot.

And it's like the latest and greatest and almost use use buzzwords as a way to attract

people.

So just, you know, be inquisitive and ask clarifying questions.

Be wary in that sense.

You are cautious when it comes to some of the buzzwords and new things and just know

what you're getting into.

Don't neglect the job hunt process.

Pay due attention to that.

Because if you just study and don't pay attention to the job hunt process, you're going to get

frustrated because you're going to finish one phase, and then have a long period of

frustration and self doubt after that.

So if you can do the job hunt phase in preparation, sort of in parallel with the latter half of

your learning journey, I think you're going to be prepared to hit the market hard when

you're ready.

And it's just a question of time after that time.

Welcome to Insiders’ Viewpoints: Advice to Women in Software Engineering.

In this video, we will hear from software engineers discussing their advice to women

in the field or who are considering getting into the field.

I think recently, there have been a lot of good positive momentum, and diversification

of the industry and making, making software engineering a more welcoming place for women

in general.

So, I come from work environments that are you know, primarily women.

And now I'm in a work environment that is primarily men.

I think I've had a pretty good experience with my coworkers personally.

But in the beginning, it did very much feel like a little bit kind of like you're alone

because you know, you're either the only or one of two or three women on your team.

I would say just remind yourself that you actually do deserve to be there that you actually

do belong.

You do belong there, and everybody started exactly where you are.

Everybody who started as an engineer and even at your company started and was like I do

not know what's going on.

But you're capable and you'll figure it out.

If you're a software developer at a company, your main job should be software development.

And if your bosses or management or other people in the company want you to do other

types of work, you should be getting compensated for it, you should have a title that reflects

what it is that you do.

And I have seen this happen a lot where the woman on the team will be the one who cares

about diversity, for instance.

And so as part of diversity workshops and clubs at the at the company and is participating

in diverse hiring events and doing all sorts of like extra training around that side, that

side of things, then review time happens.

And as a software developer, if you're getting reviewed on code, you need to make sure that

you're getting enough of that code in your life.

Everyone needs to contribute.

Everyone needs to pull their weight; it can't all go on one person.

Join a team that does make you feel like they do support women and want to elevate women

and advocate for women.

So I think it's very fair to ask directly to say, a hiring manager, you know, what,

what is your plan for diversity on your team?

What types of programs do you or the organization or company have in place to, to ensure that

woman are successful, and you know, be willing and open to speak up if there's something

that you think is uncomfortable.

It's not all on you to fix if something is broken, there are other people involved.

And they also are adults who should be reasonable human beings, and able to take advice and

constructive feedback.

So let them take that advice and constructive feedback.

You don't have to do all the changing.

never be compelled to join a team where you're not feeling good about it, or you think that

there's a risk that it's not a welcoming, so ask other people to I think it's very fair

to just ask those questions.

Don't stay, don't stay, if you don't like it, the company needs you more than you need

them almost certainly.

So, leave when you can, if something isn't working out for you leave, you have that ability.

And there are companies that have less of that in their culture and less of that in

their engineering teams.

It can feel rough trying to find it sometimes.

But if you do find yourself at a place that is not a healthy, positive, growing environment

for you leave, it doesn't reflect on you, again, it reflects on the company.

And I know, any company, or organization that's even remotely taking diversity and inclusion

seriously, simply with regards to women, will have programs in place to support women, elevate

women.

And, you know, basically, there's an eye on that, and there as there should be, there's

a lot of effort being put into hiring women and getting women into senior leadership roles,

which I think is very important.

Figure out what role you want and find free resources to get into those roles.

One thing to bear in mind is as a woman, you know, we are a minority, and we are underrepresented

in the tech industry.

So, there are a lot of free or low-cost programs tailored specifically to women who want to

get into computing or who women who are in computing looking to advance their careers.

Welcome to Job Outlook for Software Engineers.

After watching this video, you will be able to

explain why demand is high for software engineers,

describe the job outlook for a software engineer,

describe what to expect from a software engineering role

and describe the employment options in software engineering.

There are various reasons behind the current high demand for software engineers.

Almost all industries need software to compete and grow.

This results in a continuous demand for software engineers.

Most organizations require applications and websites for their company to function.

Some apps and websites are for internal use, while others are used to interact with customers.

The Internet of Things is also driving the need for software that interacts with products.

This is a need that will continue to expand for the foreseeable future.

All types of software programs, from messaging applications to commerce websites to office

software, are available in different flavors from different vendors.

Whenever one vendor releases a new version of their particular program, a competing company

is likely to also update their program,

requiring more software engineers to create the new releases, enhance functionality, and

add new capabilities.

So, the outlook is bright for software engineers.

The US Bureau of Labor Statistics predicts a higher than average 22% job growth rate

for software developers, analysts, and testers from 2020 to 2030.

That’s an average of almost 200,000 openings each year – in the United States alone!

These roles are likely to be across many industries, from mobile application development to health

and insurance.

There is also predicted to be a large increase in security software due to the increase in

computer security threats and cyber-crime.

Many software engineers are graduates with degrees in software engineering or computer

science.

However, increasingly employers like IBM, Google, and Tesla are starting to hire non-graduates

who demonstrate the required skills for the role.

This opens software engineering up to a wider range of society ensuring that the cost of

obtaining a degree doesn’t preclude candidates from the industry.

Salaries for software engineers vary significantly with years of experience in the field.

As you acquire more experience and become harder to replace, employers typically provide

higher compensation.

In the USA, salaries range from $90,000 for a junior role through to $120,000 or higher

for an experienced position.

The average of $110,000 per annum for a software engineer is more than 2.5 times the average

base salary across the USA.

This, alongside bonuses and benefits such as medical insurance, gym membership, profit

shares, and retirement plans make a software engineering role an attractive proposition.

So, what should you expect from a software engineering role?

Dress codes are unlikely to be formal unless you’re in a customer facing situation.

Hours are often flexible, sometimes around core hours set in the middle of the day.

You may occasionally be required to attend meetings or calls which can dictate a physical

or online presence at a particular time of the day.

The number of hours are likely to increase as you get near to product release dates,

often without any matching increase in pay.

The flexibility to work from home varies across employers, but there is no technological

reasons why this cannot be done.

In fact, some software teams are spread across the globe, working across all hours of the

day.

Software engineer employers include almost any type of business that you can imagine.

From large technology companies such as Facebook, Amazon, Apple, Netflix, and Google (also known

as FAANG),

to medium-sized software companies that develop specific software and solutions, to small

software houses and start-ups.

And almost all large non-technology companies like banks, retailers, and pharmaceuticals

need software engineers for their own internal or external systems.

And lastly, most employers will encourage continual learning, whether that be expanding your technical

skills or enhancing your soft skills.

They’re usually keen to invest in keeping your skills up to date so that you can develop

the best software solutions for them.

Now in the software engineering world, regardless of your job title, there are different employment

options to suit your lifestyle and situation.

Employed roles in a company or organization provide stability and a regular income.

These can range from an apprenticeship or internship, to a part-time role, through to

a full-time role.

In software engineering, more so than a typical job, there is a huge independent contracting

market and opportunities in the gig economy.

These can range from simple website development to super-specialized skills, languages, stacks,

and products.

In this type of employment, you can provide contract or consultancy services to organizations

on a time-based or project-based model

or you can work on a freelance basis taking on short-term contracts to work on projects

that interest you.

And many software engineers also code for free as volunteers contributing to open source

projects.

This can help you gain experience, enhance your skills, improve your employment opportunities,

and enhance your technical eminence.

It also makes you feel good by doing something for a cause or community or social initiative

that’s important to you.

In this video, you learned that:

the high demand for software engineers is due to

increasing needs for software,

increasing complexity of applications,

and a continuing growth of technology

the outlook for software engineers is promising

the role of a software engineer can be flexible and satisfying

and

that employment options for software engineers are flexible and varied,

from full-time employment through to voluntary activities

Welcome to Career Paths in Software Engineering.

After watching this video, you will be able to

describe typical career paths for a software engineer,

and identify the likely roles within career paths.

A career path in software engineering opens up as you gain experience and attain new skills.

Often it heads in one of two directions: technical or management.

If you thrive on working with code and solving problems, then the technical path enables

you to continue working closely with technology as your career progresses.

Or, if you exhibit strong leadership qualities alongside the soft skills of a good software

engineer, you might progress onto managing a team of software engineers and the work

that they do.

So, let’s take a look at a common career path for a software engineer progressing on

the technical or managerial path.

Remember that different organizations will use different job titles, but the roles will

be similar across the board.

You’re likely to start out in an entry-level position as a Junior Software Engineer or

Associate Software Engineer.

At this level you’ll be developing small chunks of software to meet agreed client specifications.

You’ll be assigned a team leader or mentor who will guide you along the way and provide

support when you need it.

During this phase of your career, you’ll be on a steep learning curve, gaining new

skills and experience with every block of code that you write.

You could then move on to a Software Engineer position.

In this role you’ll be expected to be more independent and able to able to break larger

tasks down into smaller achievable sub-tasks.

You may be learning new programming languages and demonstrating an understanding of the

software development lifecycle.

At this stage, you might be asked to mentor a junior software engineer or take responsibility

for a larger part of a project.

The next stage of your career could be a Senior Software Engineer position.

In this role you’re likely to have involvement across the whole of a project and with an

entire codebase.

You may be asked to mentor software engineers and to conduct code reviews across the team.

And from the experience you’ve gained in earlier roles, you’ll be expected to be

able to solve a wide range of problems in an efficient way.

If you choose to follow the technical path, then your next role might be as a Staff Software

Engineer.

In this role you’ll work as part of the technical team developing, maintaining, and

extending software.

You’ll be responsible for ensuring that the software meets customer and user expectations

and that it uses resources efficiently.

If you decide to follow a managerial career path, you could become a Technical Lead.

In this role you’ll manage the team of developers and engineers developing the software in your

organization.

You could be responsible for the entire development lifecycle and report to the project stakeholders.

Continuing along the technical path, you might progress to be a Principal Engineer or Technical

Architect.

At this stage you’ll be responsible for the architecture and design of a software

solution, as well as the development of it.

You’ll be expected to create processes and procedures for your team and provide technical

direction.

On the managerial path, you could become an Engineering Manager.

In this role, you’ll ensure that the entire team is appropriately supported and encouraged

to progress in their careers.

Becoming a Director of Engineering could be the next stage in the progression of your

career.

It is a strategic and technical role.

On the strategic side, you’ll determine priorities for the projects within the company,

identify hiring needs, and define long term goals.

On the technical side of the role, you’ll be involved in defining new projects, specifying

requirements, and overseeing the project.

The lead of an organization’s technology arm is likely to be the Chief Technology Officer

(or CTO).

In this role, you oversee all of the research and development in the company.

You’ll also monitor the company’s systems and infrastructure to ensure that it meets

your needs and budget.

Even at this level, you’ll continue learning.

As technology evolves and new products appear on the market, you’ll be responsible for

deciding which ones meet your business needs and could give your organization a competitive

advantage.

Of course, not everyone who starts out as a Junior Software Engineer follows these traditional

software engineering paths, but you’ll find that the skills you gain will benefit you

in a wide variety of other roles.

If after working on client projects for a while you decide that you prefer interacting

with the clients to writing the code, you could transition into a technical sales or

customer support role.

If you enjoy coding, but find working with numbers and data is more appealing, you could

transition into a data engineer or data scientist role.

And, if you find the data side of your new role is your passion, you could move on to

become a database administrator or database developer.

And if you enjoy finding and fixing bugs, you might transition to a software tester

role.

IT is a vast field with an abundance of opportunities across many skill sets, so starting your career

in software engineering is a great choice even if you use it as a stepping stone to

move to other technical or leadership roles in the future.

In this video, you learned that:

A career in software engineering is likely to follow a technical or managerial path.

At each stage on that path, you will take on more responsibility and a wider range of

tasks.

And, starting out as a software engineer enables you to take an array of other options at any

stage in your career.

Welcome to Software Engineering Job Titles.

After watching this video, you will be able to

list commonly used job titles for software engineers, and

identify the primary duties of and skills needed for these roles.

Job titles such as software engineer and software developer are very broad and can be used to

describe a spectrum of roles.

As well as these titles, you may come across other titles related to specific sub-domains

in software engineering.

While individual organizations might use company-specific job titles, you’ll find that most fit into

one of the more generic titles.

Types of software engineer roles include:

Front-end engineer

Back-end engineer

Full-stack engineer

DevOps engineer

Software quality assurance engineer

Software integration engineer

Software security engineer

Mobile app developer

and Games developer

Front-end engineers focus on developing the user interface, or UI, of a software solution.

They’re also sometimes referred to as UI developers or web developers.

In this type of role, you’re responsible for the visual design of the software, including

the layout of UI elements and the overall aesthetics of the application or website.

It’s important for a front-end developer to understand how users interact with software

and the principles of user experience design.

You’ll also need to understand how objects and code run differently on different operating

systems, devices, and browsers to ensure that your solution works on the user’s chosen

system.

Key skills of a front-end engineer include:

web development languages and UX and UI frameworks.

Back-end engineers focus on the business logic of a software solution.

In this type of role, you’re

responsible for the core logic of the software implementing functionality to perform tasks

such as accessing data and databases,

logging information, and caching systems by using application programming interfaces,

or APIs.

You’ll also be responsible for ensuring the scalability and performance of the solution.

Key skills of a back-end engineer include

programming languages,

application frameworks,

web servers, app servers, and load balancers,

databases.

and deployment and containerization tools and technologies.

Full-stack engineers work across the whole software solution.

In this type of role, you’re able to create

both the user interface and the back-end functionality of an application or website using the skills

of both front-end and back-end engineers.

Key skills of a full-stack engineer include

web development languages,

programming languages,

UX, UI, and backend frameworks,

web and application servers,

databases,

APIs and web services,

and deployment and containerization tools.

DevOps, or development and operations, aims to deliver software in an agile manner by

combining software development and IT operations.

DevOps engineers apply agile processes and methodologies to streamline their product

development, improvement, and maintenance

as well as transcend the boundaries of the traditionally distinct development and operations

teams.

In a DevOps role you need familiarity with both front-end and back-end technologies.

The skills, technologies, and products that a DevOps engineer is likely to use include

source code management tools,

programming languages and frameworks,

scripts, and deployment, containerization, and monitoring tools.

Depending on the actual job posting, you might use alternative products and services, but

they will be of a similar nature to those shown here.

Software quality assurance engineers, also referred to as software QA engineers or software

test engineers, test, review, assess, and write software to validate the quality of

an application.

In a QA role, you’ll develop automated tests, tools, and procedures to test the functionality

of software solutions.

You’ll use bug tracking software to log any errors you discover and report them to

the software development team.

The key skills for software QA engineers include

programming languages,

shell scripting,

bug tracking and issue management tools,

testing automation tools,

And specific software stacks depending on how the application is deployed.

Software integration engineers write code to integrate software into hardware products

enabling

smart devices and the internet of things products.

In this role, you’ll use programming languages and frameworks to program hardware such as

consumer devices, home security systems, electronics, and other interfaces.

The key skills for a software integration engineer include

programming languages and

proprietary technologies, frameworks, and toolkits.

Software security engineers, sometimes referred to as white hat or ethical hackers,

work to find security flaws and vulnerabilities in software.

As a security engineer, you’ll create systems, methods, and procedures to test software solutions

and exploit their security weaknesses so that they can be fixed before the solution ships.

The key skills for a security engineer include

programming languages,

reverse engineering,

shell scripting,

tools for vulnerability and penetration testing,

and network security and encryption tools.

Mobile app developers design, develop, and implement software solutions for mobile devices

such as smartphones and tablets.

You’ll use web skills to create front-end apps as well as learn the various different

platforms to write the back-end code interacting with the specific type of device.

The key skills of a mobile app developer include

mobile operating systems,

web development languages,

programming languages, as well as

Web services and technologies.

Games engineers write gaming software for a wide variety of devices,

from PCs, to smartphones, to web browsers, to games consoles.

As a games developer, you’ll work alongside graphic artists, sound technicians, and game

designers to create the code used in games.

In this video, you learned that

there are many different job titles under the umbrella term of software engineering,

each of which

has specific duties and responsibilities,

and requires a specific set of skills.

Welcome to Insiders’ Viewpoints: Career Pathways and Continuing Education in Software

Engineering.

In this video, we will hear from experts discussing how they became software engineers, their

education, and their background.

So I have a non traditional path into software engineering, my bachelor's degree is in social

work.

And I worked as a social worker for some time, I was introduced to coding

via social media, randomly on social media, I was scrolling in December 2018.

And afterwards, I kind of fell down the rabbit hole of learning how to program and I decided

to go to a 10 month long, very intense coding boot camp to make the switch into software

engineering.

And so one thing I have learned over the past three, three and a half years, is that your

career doesn't have to be linear, and your career doesn't have to look like anyone else's

I just picked computer science, I love math.

That's the only thing that I think really drew me to computer science was oh, it's if

you love math, apparently, you will love computer science too.

So I think some of the most successful engineers that I, that worked for me, or I've worked

with, come from backgrounds, not even sort of, like mine, neither people with like, with

non-technical backgrounds, who picked up coding on the side,

there's so many folks who come from, from you know, they are English majors or philosophy

majors or bio, whatever it is.

Because I think there's this you can you can learn all these skills and become a really

strong engineer, without having gone to a four year college or, or more for computer

science, specifically, or, or another engineering role.

And so in college, I ended up taking what was it, applied math, I did applied math as

my major.

I want to I always wanted to work on things that had a practical application, which is

why I went into applied mathematics rather than pure mathematics anyway.

Yeah, so I did that research program.

And as part of that program, we were writing simulations, using MATLAB actually.

That was a fun time.

So we were writing simulations using MATLAB, and I realized I enjoyed that part more than

I enjoyed the mathematical research part of it.

And that's kind of what kicked it all off.

Like, oh, computers are a lot of fun.

So I ended up graduating college with an Applied Math major, Computer Science minor.

And then I've been taking computer science jobs ever since then.

So I received a Bachelor of Fine Art, actually.

So not anything tech related.

And I worked in education for a few years, I ended up as an art teacher.

And it was during that time that I decided that I wanted to switch paths because I wanted

a little bit more freedom.

With my job more flexible schedule, I wanted the opportunity to work remotely.

So I decided to take a class to see if I would like it at a community college near my house.

And I really, really loved that class.

And at the same time, kind of just watch videos on Udemy and YouTube and teach myself some

things that I wasn't learning in class.

I took 32 credits at the community college level.

And then after that, I did apply to graduate school.

While in grad school, I received an internship offer at my current company and did a summer

internship and directly following that I started full time.

Next, we will hear from these experts regarding how they maintain their skills and keep up

with the continuous technology evolution.

I follow people on Twitter who talk about emerging technologies and new technologies.

I also subscribe to this newsletter called TLDR.

And they pretty much just surmise everything that's going on in the tech industry. That's what I like to do to keep up to date.

And I also take various courses on things that are interesting to me.

Yeah, I think just just carving out time to learn, whether it's within your often companies

will give people some amount of time to for personal development and growth which can

be which can mean taking courses to learn new languages or frameworks.

I've found that I learned best if I have a project that makes me need to learn the thing.

So I started with Googling, like, honestly, Reddit threads, frequently are the best places

to find out about like, what are the popular and new technologies these days.

If you just search on Google for like, the best website framework, Reddit 2022, you'll

find numerous posts of people debating the pros and cons of different types of frameworks.

And then actually, another very important thing is debug it.

Attach a debugger put in print statements.

Do everything you can to be able to inspect the flow of the code that is getting that

is getting run.

Knowing how to debug is about almost equally an important skill as knowing how to write

the code.

But the way that I kind of keep on top of everything is reading documentation.

I have really great coworkers who are always also in the know of like, when things are

updated when languages are updated when to use new new technologies.

So they helped me out a lot there.

And we also get Udemy for free, like Udemy business for free for all employees, which

is a really great way to learn new things and also stay on top of new technologies that

that come out.

Welcome to Software Engineering Code of Ethics.After watching this video, you will be able to:

State each of the eight principles in the Software Engineering Code of Ethics.

And, explain each of the eight principles in the Software Engineering Code of Ethics.

The Joint Task Force on Software Engineering Ethics and Professional Practices developed

a Code of Ethics outlining their goals and standards for software engineers engaged in

the design and creation of software.

The task force was formed by

the Institute of Electrical and Electronics Engineers Computer Society or IEEE-CS

and the Association for Computing Machinery or ACM in order to recognize the prominence

of computing in global commerce, government, and society.

This IEEE-CS ACM joint task force championed the need to hold software engineers accountable

so that the present and future status of the field as a beneficial and respected profession

is maintained.

The Code of Ethics consists of eight principles that

pertain to the specification analysis, design, development, testing, and maintenance of software,

and are dedicated to serving the public interest.

These principles are intended for anyone who is in or related to the profession, and includes

engineers, mentors, instructors, managers, students, and policymakers.

Here, we will summarize each principle.

However, the full text of each principle and its additional clarifying clauses can be found

on the IEEE-CS and ACM websites.

The principles in the software engineering code of ethics focus on the following topics:

Public

Client or employer

Product

Judgement

Management

Profession

Colleagues

Self.

The public principle comes first because it is expected that software engineers should

act primarily in accordance to positively affect the public good.

This includes accepting responsibility for their work with regard to

safety,

fairness,

accessibility, and

integrity.

Next, software engineers should act in the best interests of both the clients and their

employers.

They should act honestly and be forthright when it comes to unethical actions such as

plagiarism or illegal activities.

They should

seek consent where necessary and appropriate

and honor confidentiality.

The third principle relates to the product under production.

Software engineers should seek quality while keeping in mind cost and timelines.

This next principle outlines how software engineers act with integrity and independence

in their professional judgment.

They are expected to maintain objectivity and honesty when working with the software

and relevant documents they are involved with.

Software engineers should not

engage in any inappropriate financial activities such as bribery and double billing,

nor may they accept duties that create a conflict of interest.

Software engineering managers and leaders should also behave in a manner consistent

with these principles where it applies to them and those they manage.

They should

work to minimize risk and

employ security procedures.

Managers and leaders should also work to

ensure realistic expectations of their staff,

provide just compensation, and procure intellectual property rights whenever appropriate.

The profession is principle describes the duty of software engineers to protect the

reputation of the profession by acting with integrity and not elevating themselves at

the expense of others.

Software engineers should let managers, employers, and clients know they intend to

act in compliance with this code of ethics

and express concern over violations of the code.

Similar to the profession principle, software engineers shall treat their colleagues with

respect and fairness.

They should

encourage their peers to comply with this code of ethics and

not take credit for the work of others.

The final principle, focused on the software engineer themselves reminds software engineers

about the importance of lifelong learning and professional development.

They should

endeavor to create quality software and

strive to conduct themselves in a professional manner.

It should be noted that this code of ethics is not a replacement for conscientious decision-making

and common sense, but it can be used as a supplemental guide.

Knowing when and where to apply these principles will always be up to the discretion and wisdom

of the individual.

In this video, you learned that:

The code of ethics serves the public interest to hold software engineers accountable in

the analysis, design, development, testing, and maintenance of software

The eight principles in the code of ethics are

Public

Client or /employer

Product

Judgement

Management

Profession

Colleagues and

Self.