This document lists as bullet points the ideas of each lecture of the Ryerson Chang School course **CCPS 109 Computer Science I**, as taught and polished by Ilkka Kokkarinen over the past fifteen years. The example classes provided by the instructor fill in the language details.

Lecture 1: Classes and Objects

1.1. Classes and objects

- An **object** is an entity, existing and simulated inside the computer, that has some **capabilities** (**methods**) that the outside world can **invoke** (**call**) to ask the object to perform that particular action. For example, a bank account object might recognize the methods “deposit” and “withdraw”, but not methods “fly”, “get length” or “honk”.
- An object also contains internal **data fields** for the things that it remembers. For example, a bank account object would remember its balance and owner.
- When writing Java code, you always write **classes**, which are **blueprints** for objects. From a class that has been successfully **compiled** into Java **bytecode**, any number of structurally identical objects can then be constructed during the program execution. These objects are said to be **instances** of that class.
- An object, **once created, cannot change its type**, but as long as it exists, it will remain an instance of the class that it was originally constructed from.
- Each class should model one **concept** of the **problem domain**. As a rough rule, **nouns of the problem specification become classes, and verbs become methods**.
- When a **compiled** Java program is run inside **Java Virtual Machine (JVM)**, the execution of the program consists of objects that invoke each other’s methods.
- A stand-alone Java program normally starts its execution from the static **main method**. Using BlueJ, writing the `main` method is not needed, as you can interactively create objects and freely invoke their methods.
- Some objects constructed from the classes of the **Swing framework** have a visual presence on the user screen, and can listen to and react to the user actions done with keyboard or mouse.

1.2. Statements

- Like all programming languages, Java has a small set of possible **statements** built in the language, each one written using its own reserved **keyword**.
- Each method consists of a series of **statements** that can be written in sequence and nested inside each other to create more complex statements and **blocks**.
- Each statement must end with a **semicolon**, even if it is the only one in the method.
- In Java, nesting of statements is indicated with **curly braces**. **Indentation** is used to make things easier for human readers, but it has no effect in the language itself.
- Java is **whitespace insensitive**, except inside string literals where spaces do matter, but **case sensitive** everywhere.

- By convention, class names are capitalized, but method and variable names are not. Names that consist of multiple words are written using run-on **camelcase**, since whitespace and other special characters are not allowed inside an **identifier**.
- An **identifier**, a name in Java, can now consist of **any letter or digit, and the special characters underscore and dollar sign**. However, the identifier may not begin with digit. Note that in **Unicode**, there are *way* more "letters" than the 26 letters of English!
- Tasks that are not directly part of the the core language must either be done by writing suitable statements, or by calling a suitable method in an object that already can do that task. For example, the method `println` in the object `System.out` for console output.

1.3 Methods

- When a method is **invoked** (**called**) for some object (inside the method body, this object is referred to as `this`), the object executes the statements in the order listed in the method definition. The statements that talk about fields then access and modify the copy of the field in that particular object `this`.
- The order of statements inside a method is **highly important**, as even a tiniest change in one statement could radically change the entire program behaviour. Computer programs are the most complex and brittle of human artifacts ever created.
- The order of fields and methods listed inside the class, on the other hand, is **completely irrelevant**. Methods define what some object **can do**, not what it **will do**. The outside world decides what methods get called and in what order.
- To return a result from a non-`void` method, you must use a `return` statement. **Console output** achieves nothing here, as that is a completely separate thing.
- It is possible for class to have several **overloaded versions** of the same method. The name of each version is the same, but the parameter lists are different so that the compiler can determine based on the argument types which version is being invoked.
- Methods and statements can be **commented** with either `//` or `/* ... */` syntax. **Compiler always ignores all comments**, so they are only for the human reader to explain what is going on.
- **JavaDoc** tool to generate **class and method documentation** uses comments of the form `/** ... */` in which various **tags** such as `@param` or `@return` can be inserted. Since JavaDoc comments are still comments, they cannot possibly affect the compilation or execution.
- JavaDoc should only ever be used to document `public` methods that are **intended for other people to use**. These comments should explain only what the method achieves for its callers, and nothing about what is going on internally. Comments about private method are not intended to end up in JavaDoc pages.

1.4 Data and variables

- Variables defined inside the class are **fields** (aka **instance variables**) so that each object constructed from the class has a separate copy of each field. The fields persist inside the object as long as the object itself exists.

- As a rule, data fields should always be `private`. If outside access is desired, provide an **accessor method** (“getter”) and/or a **mutator method** (“setter”). This allows you to enforce logical constraints on the intended legal values of these methods, or make some data attributes **virtual** in that they are computed from other data that explicitly exists.
- With fields that are declared `static`, one copy is shared simultaneously by all objects. The keyword static is used for historical reasons (in programming, “static” means compile time and “dynamic” means runtime), and surely `shared` would have been a much more descriptive and accurate keyword. Too late to change now, though.
- Fields that are declared `final` may not be reassigned after **initialization**. Note that the concepts `final` and `static` are independent of each other.
- A field that is both `static` and `final` is called a **named constant**. There are many advantages of using named constants instead of writing **magic numbers** in your code.
- Variables defined inside a method are **local variables** whose lifetime spans the execution of the method, after which the local variables automatically cease to exist.
- **Method parameters** are also local variables, except that their initial values are provided by whoever calls the method. The method writer should not try to dictate them.
- Methods declared `static` are called without an underlying object `this`, prefixing the method name of the name of the class that the static method is declared in.

1.5. Types

- Java is a **strongly typed language** with **explicit typing at the compile time**. The types of all data used by the program must be explicitly declared, after which the compiler enforces that each variable is used only in ways that are meaningful to its kind of data.
- **Eight primitive types** are built in the language. In this course, we only really ever need `int` for **integers**, `double` for **decimal numbers**, `boolean` for **truth values**, and `char` for individual Unicode characters.
- The four different integer types `byte`, `short`, `int` and `long` differ by how many bytes are used to store a value. Integer arithmetic will **silently overflow** if the result doesn’t fit in, and the result is cut to fit inside the available space in a heartless Procrustean fashion.
- For humongous integers and their operations, use the class `BigInteger`.
- The compiler automatically infers the type of each **expression** from the types of its subexpressions and the operators used to combine them, thus enforcing type safety for arbitrary expressions, not just individual variables.
- Objects of type `String` are pieces of **text** consisting of a string of characters. To place a **string literal** in your code, put it inside a pair of double quotes to prevent the compiler from trying to interpret its contents in any way.
- **Escape sequences** starting with a backslash character can be used to insert otherwise unavailable special characters inside string literals. Most importantly these days, any **Unicode** character can be part of a string literal with the `\uxxxx` encoding.)
- `String` is almost the “ninth primitive type” (as in “the fifth Beatle”) since it lies closer to the JVM in many ways that the other classes do not. For example, unlike any other class (well, except one other) it has literals defined in the language level.

- Strings are **immutable**, so their contents cannot be changed after the object is created. Counterintuitively, being immutable has many surprising advantages in programming. If you need a mutable string, use `StringBuilder` instead, especially when you want to build up a string answer by repeatedly appending small pieces of text to the result.
- Any class can also be used as a type of a variable (either static, instance, parameter or local) that will then point to that type of object. New objects from the given type are created with the language operator `new`.
- Classes in the Java standard library are designed to have many useful methods that you can google as needed from the **Java API reference**.
- For each primitive type, there exists a corresponding wrapper class that contains useful static methods for that type. For example, all tests about properties of characters should be done using the static methods of `Character`. (For example, Unicode has *way* more whitespace or numerical characters than most people would assume, making all kinds of trickery and hackery possible.)

1.6. Constructors

- For reasons of both safety and security, Java language guarantees that every time a new object is created, the JVM will first fill its bytes with zeros. This guarantees that every primitive type field starts its life with value zero, `boolean` fields start as `false`, and all reference fields start as `null`.
- Occasionally you want fields to be initialized to other values. If you know this value at the compile time, you can assign it to the field at its declaration. Otherwise, write a **constructor** to the class to initialize the fields to values determined at runtime, passed to the constructor as arguments by the entity that is creating the object.
- A constructor is a special method that you never invoke explicitly. Instead, the JVM invokes it automatically on every new object that is created.
- Syntactically, a constructor has no return type, and its name must be exactly the same (including the capitalization) as the class itself.
- Constructors can be **overloaded** the same way as ordinary methods. The choice of which constructor gets executed is based on the types of the arguments given to `new`.
- **Default constructor** means the constructor that takes no parameters.
- If you don't write any constructors into your class, **the compiler automatically synthesizes in your class a do-nothing default constructor**. But as soon as you explicitly define any constructor whatsoever, this synthesization no longer takes place, and new objects can be created using only the constructors that you explicitly provide.

1.7. Assignment statement

- The most important statement for moving data around and performing computations on it is the **assignment**, for historical reasons erroneously denoted by the equals sign `=`.
- When an assignment statement is executed, its **right-hand side expression** is evaluated, and the result is copied into the **left-hand side variable**.
- Assignment is inherently **asymmetric** and would thus be much better denoted by some kind of left arrow character. Too bad that it is not.

- In **imperative programming**, each variable **contains exactly one value** at any time that that variable exists. This value persists in the bytes of memory location of that variable until a new value is assigned in that variable, irrevocably destroying the previous value.
- **Java is not a spreadsheet**. Each variable **contains only its current value**, but no **history** whatsoever of where that value came from. Changing the value of the source of some value does not change the value of the variable that the value was assigned to.

1.8. Different levels of programming errors

- **Syntax error**: the code does not conform to the absolute and hard syntax rules of the Java language. (Detected by compiler.)
- **Type error**: the code tries to do some operation on some item of data that is logically impossible, so that if that operation were allowed to compile, it would be guaranteed to crash at runtime when that statement is executed. (Detected by compiler.)
- **Runtime error**: the program crashes during execution because it tries to do something that is logically impossible. Syntax and type checking exist to prevent many of these, but literally no finite amount of such checking can possibly prevent them all at compile time. (Detected by JVM.)
- **Logic error**: the program is legal and does **something**, but **not the thing that the programmer intended**. Of the infinitely many possible programs you could write, you basically wrote some wrong one. The most difficult level of errors, and the bane of our existence as programmers. (**Detected either by you, or if you miss them, by your users**. No compiler or virtual machine could possibly read your mind to determine what your mind **intended to do**, as opposed to what your fingers actually **did type in**.)

1.9. Local variables versus data fields

- A **local variable** exists only during the execution of the method that it is declared in. The next time the method is called, the variable is created fresh. An **instance variable** (a.k.a. **data field**) inside an object exists and maintains its value as long as the object itself exists in memory.
- Instance variables are automatically initialized to zero. Local variables must be explicitly initialized in the source code before their values can be used. (This seemingly silly rule is for both safety and security in the JVM.)
- **Parameters** behave the same way as local variables, but they get their initial value from the **arguments** provided by the caller of the method. The method should never try to dictate the values of its parameter variables: if it does, that means that those variables should not have been parameters to begin with, but ordinary local variables instead.
- Instance variables have an **access specifier** that should always be `private`. For local variables, the entire concept of access specifier would be meaningless to begin with.
- Local variables exist in **stack**, whereas objects created with `new` exist in **heap**. Objects in heap don't have names (after all, it must be possible to create as many objects as we need without having to explicitly give each one of them a unique name in the source code), so we need named variables in code to be able to talk about them.

Lecture 2: Conditions

2.1. Conditional statements

- The `boolean` primitive data type has values `true` and `false` that can be used to make two-way decisions as a **condition** inside an `if-else` statement.
- The `if-else` statement executes **exactly one of its two bodies** depending on the truth value of the condition. If the else-body is empty, it can be left out altogether.
- If the body contains only one statement, the curly braces are optional, but still good style and prevent errors when the code is later modified.
- Note that the entire `if-else` is a single statement, even if its bodies consist of thousands of statements. The curly braces can therefore be left out from around that statement, but you should then be careful of the famous **dangling else problem**.
- **Multiway decisions** can be implemented by arranging suitable two-way conditions nested or sequentially into a series of two-way decisions that split the branches of some kind of **decision tree**.
- Any multiway condition can always be broken down into a series of two-way conditions in a multitude of different but logically equivalent ways. Your sense of programming style ultimately determines which way you end up writing your logic.

2.2. Comparisons and boolean connectives

- Six built-in comparisons in Java language: `==`, `!=`, `<`, `>`, `<=` and `>=`. Note that both `2 + 2` and `2 < 2` are both **expressions** in Java, the first having the type `int`, the second one having the type `boolean`. The type of an expression is automatically inferred from the types of its parts and the operators used to combine these parts.
- More complicated conditions can be created by **propositional logic** operators **and**, **or** and **not**, which are (really for no rhyme or reason) denoted by `&&`, `||` and `!`
- All the discrete math intro stuff about propositional logic is in full effect, should you happen to know some of that. Most importantly **the De Morgan's rules** for simplifying negations and double negations.
- Underneath the machine code, all computation is physically realized as complex interplay of propositional **logic gates**, complex networks of which are used inside microprocessor hardware to implement and execute **machine code instructions** that which all high level programs (even those that talk about propositional logic expressions at the high level language!) are compiled to.
- Unlike expressions in Java in general, conditions of the form `A && B` are guaranteed to be **evaluated strictly left to right**. Furthermore, these operators are guaranteed to **short-circuit** so that if `A` turns out to be `false`, the second condition `B` is not evaluated at all. This lets us simplify some expressions where `B` can potentially crash at runtime, unless **guarded** by condition `A`.

2.3. Integer arithmetic and the remainder operator

- In Java, integer division always produces an integer result, with the decimals **truncated** (as opposed to rounding to closest value). To perform a true decimal valued division, one of the operands should be a `float` or `double`.
- The `Math` utility class has good methods for **rounding**, **floor** and **ceiling**. The wrapper classes Integer and Double offer more methods for operating with these data types.
- To compute the remainder of the integer division, use the **remainder operator** `%`. (The choice of using percent sign in Java to mean integer remainder is nonsensical, but follows the historical conventions in these programming languages.)
- `a` is divisible by `b` if and only if `a % b == 0`.
- The **sign** of the result of `a % b` is always the same as the sign of `a`. The sign of `b` makes no difference to the result of this operation.
- Remainder is often erroneously called “modulus” by people who want to sound fancy, even though this is accurate only if both operands are positive.
- Assuming that `n` is not zero, which would cause the division to crash with a runtime error, it is guaranteed that `0 / n == 0` and `0 % n == 0`. So you don’t need to handle zero as a special case in division and remainder expressions. (One important difference between the mindsets of programmers versus normal people is how the former treat zero the exact same way as they would treat any other integer.)

2.4. If-else ladders

- An **if-else ladder** is an important technique to **choose exactly one** from a fixed number of possibilities. (A single if-else is just a special case of this, a little two-step ladder.)
- Some other languages have `elseif` or `elif` as a keyword, but in Java, an if-else ladder is actually a single if-else statement with an if-else ladder nested inside its else-branch.
- Removing the braces and adjusting the indentation makes the structure look like a downwards ladder that consists of a series of individual steps, hence the name.
- When the execution of a method arrives at an if-else ladder, the conditions of its steps are evaluated in order. Once some `true` condition is found, the corresponding body is executed, and the rest of the ladder is skipped altogether. Even if there are more conditions that are true, they will not get executed.
- The order in which the steps are listed in the ladder is therefore absolutely essential. **Changing this order will change the logic implemented by the ladder.**
- An if-else ladder whose last step is not an unconditional `else` is perfectly legal, but seems iffy (heh) and is probably somehow wrong. (This is not an absolute rule, though.)
- When writing an if-else ladder, you can safely assume in each step that all previous conditions are false. This may allow you to simplify the conditions, especially if you arrange these steps in a clever way to begin with.

2.5. Ternary selection and switch

- Java makes a distinction between **statements** and **expressions**. Statements **do something** (that is, after executing a statement *something* can be different from the way

it was before that statement), and may or may not evaluate to a value. Expressions **evaluate** to a value, but don't change anything.
- Expressions are used as building blocks for statements that then use their values.
- **Ternary selection** of the form `cond ? expr1: expr2` behaves just like an if-else, but **is an expression**, not a statement, and evaluates to either `expr1` or `expr2` depending on the truth value of `cond`.
- Ternary selection can occasionally be a handy way to sneak some small decision (for example, whether to pluralize some word) inside some larger unconditional statement such as a method call or output statement.
- `switch` in Java and related languages is just moronic, so let us simply ignore it and use an if-else ladder instead. Some other higher-level languages (e.g. Ruby, Swift, Wolfram) have a more useful switch control structure.
- In a `switch` statement, each `case` has to be a compile time literal whose type is either `int`, `String` or some `enum`. In such a situation, `switch` can theoretically be more efficient than an if-else ladder, provided that a sufficiently smart compiler can turn its logic into a **jump table** that allows the execution to jump directly to the correct case without having to examine all the cases before it.

Lecture 3: Loops

3.1. General repetition

- Some methods need to be able to do something a number of times that cannot possibly be known at the compile time, and thus cannot be hardcoded into the method.
- Furthermore, if the number of statements in a program is set in stone at compilation, the only possible way to dynamically execute a very large, even unlimited, number of statements is to execute the same statements over and over as many times as needed.
- In **structured programming**, such repetition is expressed by grouping statements into various **loops** whose **bodies** are **blocks of statements** to be repeated.
- Before structured programming in the seventies and earlier (and even today in low-level machine code), the control of execution used to consist of jumping back and forth within the code with the **goto statements** famously considered harmful. For nontrivial programs, this sort of tomfoolery produces unstructured **spaghetti code** whose logic could not be understood or modified in any safe or controlled fashion.

3.2. While-loops

- A **while-loop** consists of a **condition** and a **body**. When the execution arrives at the while-loop, the condition is evaluated. If it is `true`, the body is executed once.
- After executing the body, the execution jumps back to the beginning of the while-loop to evaluate the condition and execute the body if that condition is still true.
- Note that the condition of the while-loop is tested only between the rounds of the body, and therefore can be temporarily `false` inside the body without terminating the loop.

- A while-loop can execute its body (1) zero times (the condition is `false` to begin with), (2) a finite nonzero number of times (the condition is initially `true` but will eventually become `false`), or (3) infinitely long (the condition will always be `true` no matter what).
- In this course, **infinite loops** are a logic error. They usually result from the body of the loop containing multiple branches, some of which you forgot to make do something that advances the loop.

3.3. Achieving practical goals with while-loops

- The intention behind all **while-loop** to achieve goals is based on the simple principle of "**While you have not yet reached your goal, take one more step towards it.**"
- The infinite-way decision of how many times the body of the loop should be executed is broken down into a sequence of far simpler two-way decisions "Does the body of the loop need to be executed for one more time?"
- For some problems, we can actually know the number of rounds at compile time, but still use a loop instead of repeating the same statements in the source code. For some problems, we can't know this number at the compile time, but we can reason how many rounds are needed the moment the execution arrives at the loop at the runtime.
- Such problems are often solved by maintaining a **loop counter**, a local integer variable that keeps track of how many rounds have been executed, or equivalently, how many rounds still remain to go.
- However, there exist problems for which the number of rounds required to reach the goal cannot be known beforehand even in principle, and the only way to find it out is to execute the while-loop and see what happens. The while-loop has no problem with this, as long as it is possible to determine whether you have reached your goal.
- The goal can always be zero steps, *n* steps or infinite number of steps away. You just have to keep going to find out.
- It is important that the while-loop can terminate immediately whenever it turns out that you were already standing at the goal to begin with! **When there is nothing to do, the correct thing to do is to do nothing**.

3.4. Do-while loops

- Even though while-loops are computationally universal, in some situations `do-while` and `for`-loops can be more practical and convenient. (There is no efficiency difference, since everything becomes the same simple machine code comparisons and jumps anyway.)
- Do-while loop is guaranteed to execute its body at least once before even looking at the condition. After this unconditionally forced first round, the do-while loop behaves **exactly like** the while-loop.
- Do-while loops tend to be rather rare in practice, but can come handy in situations where the condition does not really make sense until one round of body has been executed. Typical example is reading user input until the user provides a legal value.
- A dead giveaway that you should be using a do-while-loop instead of while is having to repeat either some or the most of the body of the loop before the loop itself.

- More generally, **always let the computer do your repeating for you**. That is what it is there for! Whenever you feel the urge to repeat yourself in writing the code, **you are almost surely doing something wrong**. It is fine to copy-paste from Stack Overflow and such, but whenever you copy-paste some chunk of your own code inside the same project, stop that and go back to the drawing board and try to think in a higher level of abstraction.
- "*Three or more, use a for!*" as the famous programmer chant says. (Or, extract the common code into a separate method and call that.)

3.5. For-loops for iterating through ranges

- **For-loops** are designed for the common task of iterating through **a sequence of values** (typically integers) and **processing these values one at the time** by executing the loop body once for each value in the sequence.
- An integer sequence is defined by (1) its starting value, (2) where it ends and (3) the step size between two consecutive values. Not coincidentally, the definition of for-loop contains three parts before the actual loop body, to define these exact three things.
- However, instead of defining where the sequence ends, you specify the opposite, whether you are still inside the sequence. The **condition** of the for-loop determines whether the loop counter value is still inside the sequence.
- Be careful not to have any **off-by-one errors** in the condition, so that the last value of the sequence is also correctly handled, and that the loop does not attempt to access the nonexistent value after the intended sequence.
- If the condition is `false` to begin with, the sequence is empty and there is nothing to process. Again: when there is nothing to do, the correct thing to do is to do nothing!
- Inside the body of the for-loop, the loop counter variable is available to determine which value of the sequence the loop is currently at.
- Some loop bodies just ignore the loop counter value, since the loop counter is only needed to keep track of the count of the operations that work the same way regardless of the current value of the loop counter.

3.6. Nested loops

- Since the body of the loop can consist of arbitrary statements, and loops themselves are also statements, not only is it possible but perfectly legal and good to occasionally **nest** an **inner** loop inside an **outer** loop.
- As always, there is no limit how deep such nesting could go. As the famous zero-one-infinity principle of computer programming says, **the moment you allow there to be two of something, you really should allow any finite number of that something**. This simple principle pops up its head all over computer programming and software engineering in design, programming and execution.
- (Yes, this principle itself is misnamed: it really should be *zero-one-**arbitrary***. Sigh.)
- The behaviour of the nested loops might not initially be that intuitive to grasp. You should especially note that the inner loop is **executed in its entirety from the beginning every time** the outer loop moves forward one notch.

- To get an intuitive idea, think of the **behaviour of an ordinary clock** that measures hours, minutes and seconds, implemented as three nested loops.
- The outermost loop counts hours from 0 to 23. For each such hour, the inner loop for minutes would go through its entire range from 0 to 59. For each such minute, the innermost loop would go through its entire range from 0 to 59.
- In some other nested loops, the inner loop will run for a different number of rounds each time depending on the value of its outer loop counter.

3.7. Additional goodies for working with loops

- Since the loop counter update is most of the time either a simple **increment** or **decrement** by one, Java offers shorthand operators ++ and `--` for that purpose.
- Both operators can be used to any integer variable from left or right. Used from left, the variable is incremented before use; used from right, after use.
- A variable that is incremented or decremented inside a statement should not appear **anywhere else** inside the same statement, since otherwise the **entire statement becomes undefined**, although it compiles fine, and could even run without any errors if you are lucky.
- With modern compilers, there is no speed difference between `i++`, `i=i+1` or `i+=1`.
- To terminate the loop on the spot, use the `break` statement. This can be handy if you somehow realize that you are done and it is pointless to execute the rest of the loop (for example, you are using the loop to look for just one value that satisfies some existential claim, or one counterexample to a universal claim).
- To skip the rest of the loop body and jump directly to the next round (but not terminate the loop entirely), use the `continue` statement. This can be handy if you want to skip the processing of some special values inside the sequence.
- (The `continue` operation is called that for historical and technical reasons: far more descriptive names would have been either `skip` or `next`. Again, sigh.)
- Both `break` and `continue` only jump out of the innermost loop that the execution is currently in. If you want to jump out of an entire nested loop structure, give it a **label** and then use that label in the statement.

Lecture 4: Swing

4.1. AWT and Swing

- From the first version onwards, Java was shipped with the **Abstract Window Toolkit** standard library for creating simple interactive GUI components.
- AWT is essentially a set of **hooks** to the native GUI components from the underlying window system. As a result, the exact same Java programs and applets had a different behaviour and look and feel in different environments, contrary to the universal spirit of the language expressed in the Java marketing slogan "Write once, run anywhere".
- Furthermore, Java GUI’s were restricted to the least common denominator of GUI components that all window systems had in common.

- This and various other deficiencies inspired the later creation of **Swing** that renders everything by itself to achieve a consistent look and behaviour through different systems.
- Except for the completely redesigned component classes themselves, **Swing reuses and extends most of AWT**, which is why many `java.awt` packages get imported whenever you use Swing.
- To avoid naming confusion between Swing and AWT component classes, all Swing component classes are named with a **capital J in front of the name**, to emphasize that that class is a Swing component whose objects have a visual presence on the screen.

4.2. Creating your own Swing components

- To create a component class of your own, **extend** any existing Swing component class (in all our examples, we extend `JPanel`) to **inherit** all the Swing functionality for free into your own **subclass**.
- (This is a sneak preview for the course CCPS 209 where the theory and practice of **inheritance** will be thoroughly investigated, but we can already get a nice smooth taste of it with Swing, without needing the full underlying theory. Historically, GUI programming was the original "killer app" of object oriented programming, showing the power and flexibility of OOP techniques compared to traditional imperative programming.)
- In your subclass, you can then write new methods and **override** inherited methods for additional functionality that the original component does not have.
- Swing components have many possible properties and tweaks for their behaviour. Most of them have reasonable default values, but the one property that Swing can't possibly know or infer about your component is its **preferred size**. You should always set this property in the constructor.
- Other possible properties include the background and text colour, decorative borders, font, accessibility, localization, tooltips…

4.3. Rendering your component

- At any time, Swing can decide that your component needs to be rendered. Every Swing component class has a special method `public void paintComponent(Graphics g)` that performs the rendering on command.
- Nothing is guaranteed about how often and when Swing will call this method. Your component must be ready for it at any time.
- After this method has finished drawing the component on the `Graphics` object (actually, these days a far more powerful subtype `Graphics2D` that is much better suited for the rendering expectations of this millennium), Swing posts this image on the user screen.
- Various **geometric shapes** from the library `java.awt.geom` can be drawn on the surface of a Swing component. To **draw** or **fill** a geometric shape using the given pen and colour, use the methods `draw` and `fill`.
- The coordinate system of each component has the **origin** (0, 0) at the **top left corner**, not bottom left like it was in your math classes. **The y-coordinate then grows downwards**, which you should remember when positioning your geometric shapes.

- The units of the coordinate system correspond to the individual pixels of the component. However, the shapes are still defined using the `double` precision, which affects **antialiasing**.
- (For all graphics and especially animations, the **motion paths** should be computed in `double` precision even if the individual frames are rendered into integer pixel coordinates. Otherwise these paths will look ugly, especially with slow motion.)
- Text can be rendered using the method `drawString`, given a `String` and the starting point of the text **baseline**.
- Colours are defined as objects of class `Color`. Each possible colour is defined by its **RGB components**, each of these an **unsigned byte** with a value from 0 to 255. The utility class `ColorSpace` has methods for converting RGB colours to and from other colour spaces more intuitive to humans.

4.4. Adding components inside other components

- Swing components can also contain other components inside them. To add a component inside another one, simply use the method `add`.
- The component classes `JFrame`, `JDialog` and `JApplet` (and all their subclasses) are **top-level containers** that have the ability of existing on the user screen on their own. All other Swing components must be placed inside some top-level container for them to be able to show up on the screen.
- (The Swing component objects are still regular objects that exist in the heap the same way that all objects do. The top-level container component is needed only to make these components visible to the user on the screen.)
- When you add a component inside another component, you typically don't want to use **free positioning** for its coordinates and size the way you do with graphical shapes. Instead, you let the outer component's **layout manager** do this positioning for you, to spare you from this grunt work especially with components that can be resized.
- Layout manager is a puppeteer working behind the scenes that positions components according to its built-in rules. It is not a visible Swing component of its own. The simplest layout manager used in our examples is `FlowLayout`. More advanced useful layout managers are `BoxLayout` and `GridLayout`.
- Arbitrarily complex GUI layout can be created by nesting Swing components inside each other, possibly using different layout managers in different places.
- All layout managers try to respect the preferred sizes of the components, but this is not always possible if you jam too many components in one place.
- To perform free positioning of your components, use `setLayout(null);` and then for each component you add, explicitly call its methods `setPosition` and `setSize`.

4.5. Event handling

- All Swing components automatically generate **events** to announce that something happened to them, such as the user clicking them with a mouse. There are many different subtypes of events for various things that can happen to a component.

- If nobody has **registered to listen** to certain types of events for a particular component, those events will just vanish since nobody cared about them. In fact, most events will simply be ignored since there are just too many of them to care.
- Once an **event listener object** has been registered for that component, the method of that object that corresponds to that particular event gets automatically called by Swing. You should write the body of this method to do whatever it is that you want to happen to react to that event.
- For example, the `MouseListener` event handler has five different methods: button down, button release, button press, mouse enters component, mouse exits component.
- Each event handling method receives as argument an **event object** that contains all available information about the event that took place. For example, for a mouse button being clicked, the pixel coordinates of that click, which button it was, and whether it was a single click, double click or triple click.
- To ask Swing to repaint your component since the way that it looks is supposed to change as a reaction to some event, call the method `repaint` to place this request into Swing's internal queue. Never call the `paintComponent` method on your own.
- All Swing component classes such as `JLabel` already know to request repainting on themselves when something changes, so you don't need to do that for them.

4.6. Nested classes

- Event handlers for a Swing component class are usually best written as `private` **nested classes**. The event handling functionality should not be part of the `public` interface of the class. (Yes, many examples that you find on the web do exactly that, even though this is false advertising at best.)
- In Java, when some class is nested inside an **outer class**, the objects of this nested class cannot exist on their own, but must always exist in the **context** of some object from the outer class. This is the defining criteria for deciding whether some class should be implemented as a nested class or a separate top-level class of its own.
- It follows that these nested class objects can only be created inside the instance methods of the outer class. The object `this` becomes this context object.
- In the heap memory, each object created from the nested class contains an unnamed reference into its context object. JVM can follow this reference to access the context object.
- Since they are defined inside the outer class, the nested class methods can automatically read and write **all members** of the outer class, even those that are `private`. No special syntax is needed, since the methods of the nested class can simply talk about the outer class members as if they were their own.
- To resolve an ambiguity between a local variable `x`, a nested class field `x` and an outer class field `x`, say either `x`, `this.x` and `A.this.x`, where `A` is the name of the outer class. (Of course, you could simply just not use the same variable name in both outer and inner classes to avoid this headache in the first place!)
- If some nested class is only ever used in one method of the outer class, it can be placed inside that method as a **local class**. A local class has the same privileges as a nested

class, but can additionally access the `final` local variables of that method. (The technical explanation for this restriction to `final` is quite complex. I will give it in 209.)

Lecture 5: Arrays

5.1. Motivation for arrays

- So far, each variable has been a **scalar** that contains exactly one value at the time. Programs that need to store large amounts of data simultaneously would then need to declare a large number of scalar variables in the program source code.
- An **array** is a handy way to declare and store **an arbitrary number of scalar variables under a single name**, all done in a single declaration.
- For (almost) any legal type `T` in Java, there exists the corresponding **array type** `T[]` whose individual element type is `T`. ("Almost", since `T` cannot be any **generic** type, even if fully instantiated. See below. The technical reasons for this are rather dumb, see 209.)
- Java arrays are **homogeneous**, so each element inside the same array is always of the same type. This allows the compiler to enforce type safety at compile time same as it does for scalar variables.
- Even if `T` is one of the eight primitive types, the array type `T[]` is a class. Therefore each variable of that array type is a **reference** pointing to the array object stored in the heap.
- The number of elements stored in the array object does not need to be known at the compile time. However, this **length** will be set in stone at the time the array object is created. At runtime, arrays are **immutable in length**, but **mutable in content**.
- If the intended length is known at the compile time, the array elements can be **initialized** by listing them in curly braces. This can only be done at the time of array declaration. (This is also how you enter an array as a method argument in the BlueJ interactive method calls.)
- When an array (or any object) is passed as an argument to a method, it is **passed by reference**, with only the memory address given to the method. It therefore takes the same amount of time to pass or return a ten-element array as it would take to pass a giant billion-element array.
- The modifications to the array done by the method persist to the caller, since both the caller and the method see and share the exact same array object in the heap.

5.2. Indexing

- To access the element in the given position `idx` in the array `a`, use the square brackets **indexing** operator `a[idx]`. Regardless of the element type, the array index is always an `int`. (One array object can therefore have at most about two billion elements, even if your heap memory had room for more elements.)
- The positions inside the array `a` are numbered from `0` to `a.length-1`. Trying to use any index that is **out of bounds** is **guaranteed to crash the method at runtime**. This guarantee made by the JVM immediately prevents many **memory corruption** bugs and **security holes** in programs that would otherwise be rather difficult to find and debug.

- Zero-based counting can first be a bit alien to people accustomed to one-based counting, but zero-based counting does have several advantages in programming.
- Arrays are a **random access data structure**. At any time, **you can access the element in any position in the same constant time**. Often, a for-loop is still the most convenient way to process the array elements **sequentially** one at the time.
- "Random access" would be more accurately called "arbitrary access", but the term is already far too established. Even worse, the distinction between **RAM and ROM memory** uses the very same term "random access" to mean that the stored values are **mutable**! (Facepalm.)
- If the order of processing the elements is immaterial for the problem, the easiest way to go through these elements is to loop from beginning to end. For this common operation, you can use the alternative, more concise syntax of a **for-each loop**.
- **An array can be empty, with its length equal to zero**. Any attempt to index such an array is automatically out of bounds. Empty arrays are not an error, but a perfectly meaningful and necessary possibility for methods that return arrays as answers whenever there exist no elements that satisfy the conditions of the problem.
- Note the difference between the empty array and `null`, the latter meaning that there is no array at all. Return an empty array when you have no elements to return, not `null`.
- Both kinds of for-loops **correctly do nothing** when used on an empty array.

5.3. Array utilities

- Since all array types in Java are classes, there is no law of either nature or man that would have prevented these classes from having useful methods. However, in Java, **arrays have diddly squat**. This is a massive design flaw in the language. We know better now, and should have known better even in the angst-filled nineties.
- The standard library class `java.util.Arrays` contains `static` utility methods for common array operations such as **copying, subarray slicing, sorting and searching** and other common operations that really should have been methods in the arrays themselves.
- Especially the `equals` and `toString` methods that by law have to be in the arrays themselves are flat out useless, and you therefore want to always use the corresponding methods provided by the `Arrays` utility class.
- In the class `System`, the static method `arraycopy` is used to copy an arbitrary subarray of the source array into any position of the target array. (This **native** method performs the memory bulk copy directly on the computer hardware for maximum speed.)

5.4. Useful tidbits

- The Java language standard guarantees, both as a **safety** and **security** feature, that whenever any `new` object is created, the very first thing that happens to it is **filling the memory bytes with zeros**. For arrays whose element type is one of the eight primitive types of Java, **all elements are thus guaranteed to be initialized to zero**.

- If the element type is some class, its elements are references to such objects, and are automatically initialized to `null`. Therefore when creating such an array, remember to also create and fill its actual elements, not just the empty references to them.
- A `boolean` array is stored in memory as a **bit vector**, packing eight `boolean` elements into each individual **byte**, one **bit** per element.
- A common pattern in creating and filling an array based on **dynamically generated values** is to maintain an index (in all my examples named `loc`) of the next available location. Whenever a new element is added to this location, the variable is incremented afterwards so that the next element to be added will go to the next location.

5.5. Arraylists

- `ArrayList<E>` is a useful **collection class** that acts as a higher-level array that can grow as needed, optimized for the common operation of adding elements to the end.
- `ArrayList<E>` is a **generic class** whose element type `E` has been left open. When actually used, the **type argument instantiation** is given between angle brackets, for example `ArrayList<String>` or `ArrayList<Integer>`. All these type instantiations can live together inside the same program, and they are different types that are not **assignment compatible** with each other.
- The eight primitive types cannot be used as type arguments, but the **wrapper classes** can be used in their place. Java can automatically convert between primitives and the corresponding **wrapper types** with its internal **autoboxing** mechanism.
- Internally, each `ArrayList` object uses a primitive array with additional **slack** in the end, internally allocating a new bigger array when the object runs out of slack space. (The class `StringBuilder` operates internally using this same principle.)
- More generally, the **tradeoff between memory use and running time** is well known in computer science, showing up all over the place in design of programs and algorithms. Since memory these days is almost too cheap to meter whereas time is money, we are happy to trade more memory for less time whenever we can.
- Generally, the idea is to remember the results of earlier computations so that you can simply look them up later as needed instead of recomputing them from scratch.
- "*All programming is an exercise in caching.*" (Terje Mathisen) "*Caching leads to cha-ching!*" (Ilkka's corollary)
- Every arraylist object starts out its life as being empty, although you can pass the intended capacity as the constructor argument, to prevent the need for future expansions if you know how big your arraylist is roughly going to become.
- Use the method `add` to dynamically add new elements at its end, making the arraylist size grow by one in the process. As long as there is slack space, adding to the end is fast.
- To find out the present length of the arraylist, use the method `size`. (Arraylist is just one of the **Collection Framework** classes, some of which have no concept of “length”, but every collection has the concept of size.)
- The methods `get` and `set` are used to access the element in the given position.
- The for-each loop works for arraylists and all collections just as it works for arrays.

- To convert an arraylist `al` whose element type is `T` to an ordinary array of type `T[]`, use the method call `al.toArray(new T[0])` with the dummy argument telling the compiler the result type. To convert an array `ar` to an arraylist, use `Arrays.asList(ar)`.
- Arraylists have good methods built in, especially they have decent `toString` and `equals` methods. For additional static utility methods, see the utility class `Collections`.

5.6. Multidimensional arrays

- **Java really has only one-dimensional arrays.** However, since arrays can be used as the element type, a **two-dimensional array** is an array whose each element is an array. In practice, this behaves exactly as a true **two-dimensional grid** would behave.
- For a two-dimensional array, the expression `a.length` gives the number of rows in it.
- Two-dimensional arrays can be **ragged**, that is, each of its rows can have a different length. To find out the length of a given `row` of array `a`, use `a[row].length`.
- To access an element in a 2-D array, use **two pairs of square brackets** to index the position. To access an individual row as a 1-D array, use one pair of square brackets.
- To loop through the elements of a 2-D array, use **two nested for-loops**. The outer loop goes through the rows, and for each row, the inner loop goes through the positions in that row.
- A three-dimensional array is an array whose each element is a two-dimensional array.
- Theoretically, there is no limit how many dimensions an array could have, so the type system offers an infinite staircase of array types of ever higher dimensions. However, in practice one rarely sees more than two dimensions, and virtually never more than three.

Lecture 9: Algorithms

9.1. Computational problems

- A **computational problem** consists of the **domain** of possible **instances** and their expected correct **answers**. The domain determines what kind of arguments the problem is meaningfully defined for.
- Giving values to the **placeholder variables** of the problem gives you one possible instance of that problem. The problem itself does not have an answer, but each one of its instances does have a definite answer.
- For example, one possible instance of the famous **primality** problem “Is the integer *n* a prime number?” would be “Is the integer 17 a prime number?”
- A problem where the answer is always either true or false is called a **decision problem**. All computational problems can in principle be reduced to a series of decision problems.
- Every interesting computational problem has an **infinite number** of possible instances. Otherwise, that problem could be trivially solved with a finite **lookup table**. (However, some problems are still very interesting because even though this finite lookup table must necessarily exist, we humans do not know the correct way to fill it!)

9.2. Algorithms

- An **algorithm** is an **unambiguous and deterministic series of instructions** whose execution is guaranteed to produce the correct answer for any particular instance in finite time (that may depend on the particular instance).
- For example, the simple **brute force algorithm** for primality that checks all possible divisors of n, going up to the square root of n, still a pretty big number for nontrivial n.
- The individual instructions of the algorithm must be such that the entity executing the algorithm is able to perform them, be that entity a computer or a human.
- An algorithm is one level of abstraction above any actual computer program or method that **implements** it. The same algorithm can be implemented in different programming languages without changing the essence of that algorithm.
- Executing the steps of the algorithm must be sufficient to solve the problem, no other intelligence or decision-making is needed. This allows **computers** to exist, machines that perform computations without any understanding of what they are doing and why.

9.3. Asymptotic running time

- The running times of algorithms are expressed in **asymptotic notation** $O(f(n))$ as a function of the input size n. This is because actual running time as measured by the wall **clock** depends also on the speed of the computer that is used to execute the program.
- When calculating the asymptotic running time of an algorithm as a function of n, consider only the highest order term, and ignore the constant coefficients.
- Infinitely many algorithms and running times reduce to the same asymptotic running time, so is there anything useful in knowing that, say, some algorithm runs in time $O(n)$?
- Why yes, there is! If algorithm A is known to be **asymptotically faster** than algorithm B, it is **mathematically guaranteed** to also be **absolutely faster** from some input size n upwards. The asymptotic notation maintains just the right amount of information to allow comparisons of the running times of competing algorithms, while abstracting away the irrelevant details that would only complicate reasoning analysis.
- Because of the constant factors that are abstracted away, sophistication overhead might still lose to simple **brute force** when n is small.
- The asymptotic running time can usually be reasoned from the algorithm structure without actually implementing or measuring it. This is easy because all constant factors can be considered to equal 1, a very easy number to perform computations with.
- The asymptotic running time of a fixed-length sequence of operations is the maximum of the individual steps. For algorithms consisting of nested loops, calculate the asymptotic running time inside out.

9.4. Array searching

- There can exist algorithms of wildly different asymptotic running times for the same computational problem. The **inherent complexity** of that problem determines how far down we could theoretically push these limits, but no further.
- For many interesting problems (e.g. **matrix multiplication** or simply the elementary school algorithm for **integer multiplication digit by digit**) their inherent complexity is

currently unknown to even Her Majesty's finest scientists, and the question of fastest algorithm is still an open research problem.

- The computational problem “Does the array *a* contain the element *x* somewhere?” has the inherent complexity O(*n*) if nothing is known about the elements and their order. This is because no algorithm can answer *false* without looking at every element once.
- The one-pass **linear search algorithm** loops through the elements in linear order, working in linear time.
- Using the **parallel computers** of the future, some problems may **parallelize** efficiently to break the limit of their inherent complexity. Some other problems, though, can be proven to be **inherently sequential** and thus will not parallelize at all.

9.5. Micro-optimizations

- Even after an asymptotically optimal algorithm for some problem has been discovered, we might still want to speed it up by a constant factor by shaving down the coefficient of the highest order term with **micro-optimizations** of rearranging code and data locally to make execution faster.
- Linear search offers good examples of two famous micro-optimization techniques of **sentinels** and **loop unrolling**.
- In practical programming, **micro-optimizations are usually not worth doing** in the high level language source code, for four reasons: (1) Compilers already do most of them (2) Code becomes more inflexible and harder to understand and maintain (3) The intended optimization doesn’t speed up anything due to the discrepancy between the mental model of execution in the high level language and the reality of machine code in the physical hardware, so the intuitions of the former don’t apply to the latter (4) The optimized part won’t actually be executed that often anyway, so it makes no discernible difference to the total running time even if it were magically optimized to zero time.
- Before making it fast, make it clear and correct. There is no point optimizing code that doesn’t work correctly to begin with.
- Before any optimization work is done, the code should first be **profiled** to find out where the actual **bottlenecks** and **hot spots** reside, and concentrate all optimization on them.
- **One algorithmic optimization will always beat any number of micro-optimizations**.
- Programmer time and limited brain cycles are also valuable, especially in programs that will not be run that often.
- The nonexistent parts of any system can't break down, take execution time, increase the total complexity, or cause any other problems. In general, nonexistence is a highly underappreciated phenomenon in all walks of life, not only programming.
- “*Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time:* ***premature optimization is the root of all evil.*** *Yet we should not pass up our opportunities in that critical 3%.*” (Donald E. Knuth)

9.6. Binary search

- If the elements of an array are **known to be in sorted order**, searching for an element can be done massively faster than for an unsorted array by using **binary search**.
- As everyone intuitively understands, it is much easier to find what you are looking for when your data is sorted, which is why public libraries don't just toss their books randomly on the shelves, and buildings are numbered incrementally instead of randomly along the street they are on.
- Binary search is a classic algorithm and programming exercise that turns out to be surprisingly tricksy to avoid getting stuck in an infinite loop: reputedly, it took over a decade for the first correct implementation to be published after the first publication.
- To search for element x in the given subarray, compare it with the **midpoint element** of that subarray. The result allows you to **continue from only one of the halves** of that subarray, and discard the other half completely.
- The idea of binary search generalizes to solve many other similar problems, for example finding some root of the given continuous function that we have a black box to evaluate at an arbitrary point, but need to know nothing more about that function.
- Since every comparison cuts the number of remaining elements left to consider to about half, the running time of binary search is **logarithmic** O(log n) and thus very fast even if n were astronomically huge.
- In computer science, **base two logarithms** are more convenient than usual base e or base 10. (Asymptotically, all logarithms are equal within a constant factor, so just pick whichever base is most convenient for the situation at hand.)
- Conversion between base 2 and base 10 powers is done easiest with the handy approximation $2^{10} = 10^3$. For example, one billion, which equals 10^9, is roughly 2^{30}. If you have a haystack of one billion possibilities, to find one little needle hidden somewhere in it, you have to cut that haystack in half only about 30 times.
- A haystack whose size is the entire universe (about 10^{80} elementary particles, give or take a couple of orders of magnitude), needs to be cut in half about 250 times to find one particular (heh) particle. **Repeated halving** could therefore effortlessly bring down problems whose search space is literally as big as the entire universe!

9.7. Families of asymptotic running time

- O(1) **Constant time**, regardless of array size n. **Hash table** operations on average.
- O(log n) **Logarithmic**, could almost just as well be treated as a constant for most practical purposes. **Repeated halving** such as binary search or binary exponentiation.
- O(n) **Linear**. **One-pass algorithms** through the array that always go forward, never retreating (or at least retreating only a constant number of steps per element, aggregated over all the elements of the array).
- O(n log n). Typical of algorithms that consist of a single pass followed by two **recursive calls** for the two halves of the arrays. Quicksort, mergesort.
- O(n^2). **Quadratic**. Two nested loops to process an array. The difference between quadratic and linear is **the most important hurdle in practical algorithmics**. **The Poor Shlemiel tale** is a nice recent illustration of this phenomenon.

- $O(n^3)$ **Cubic**. Three nested loops. Highest order of polynomial that ever occurs in practice. (Some exceptions to this in theoretical computer science.)
- **Quartic**, **quintic**, and so on, **don't really exist in practice**. Most people will rather give up than write that many loops nested inside each other. Already infeasible for large n.
- $O(2^n)$. **Exponential**. Many **combinatorial** entities such as **subsets** and **permutations**. Utterly infeasible even for the fastest parallelized supercomputers, except for small n.
- (Suppose you are writing, say, some card game that uses hands made out of exactly five random cards out of 52, so you know that n will always be exactly 5. Now you have no problem going through all subsets, combinations or permutations of the given hand.)

9.8. Sorting

- **Sorting an array** (to ascending order, as the convention says) is an important classic problem in computer science with many algorithms and techniques of highly educational general value. These days sorting is a nearly solved problem with the improvements only shaving down the third decimal place… or is it?
- Of the simple $O(n^2)$ algorithms, **insertion sort** is the best, in fact **provably optimal** among comparison sorting algorithms that are allowed to **compare and swap only adjacent elements**.
- Other quadratic algorithms **selection sort**, **bubble sort** etc. perform redundant comparisons along the way, and thus are not as efficient as insertion sort.
- Insertion sort has additional advantages such as being **online**, and working fast for arrays that are “almost sorted”, for many reasonable definitions of such.
- To break the quadratic bound and reach the much better $O(n \log n)$ that is known to be the inherent lower bound for all **comparison sorts**, the algorithm must be able to compare and swap elements located arbitrarily far from each other in the array.
- **Merge sort** from the fifties, that smoke-filled decade when the nation's Top Men with horn-rimmed glasses and buzz cuts built the first room-sized clunkers for mechanical computation, is handy for sequential data that is read and written in streams. To sort an array, split it right in the middle into two halves. Sort these halves recursively, and then merge the sorted halves together in a single pass merge.
- **Quicksort** from the seventies is "sort of" similar to merge sort, but turned “upside down”. To sort an array, first **partition** its elements to “small” and “large” subarrays comparing them to some arbitrarily chosen **pivot** element, then recursively sort both subarrays.
- The choice of pivot determines how quickly the sorting works. With a bad choice all the time, the algorithm can degenerate into an inefficient recursive version of selection sort.
- Quicksort is also a rather tricky algorithm to implement correctly, but it offers many places of fun optimization and clever hackery. Mergesort has a lot less freedom of motion to dance around.
- **Dual pivot quicksort** is a recent invention that speeds up quicksort by about 10%. Sorting was long thought to be a solved problem with only that "shaving down the third decimal place" left to do for researchers, but surprisingly that was not so.

9.9. Data structures

- A **data structure** is some way to store and organize data so that some operations on that data become faster compared to keeping that data in an unsorted array.
- Different data structures make different operations faster. Choose the data structure that speeds up the operations that you intend to be doing a lot, so the others do not matter.
- Classic data structures are already implemented as libraries in all serious programming languages, so there is rarely a need to reinvent the wheel.
- A data structure is called a **dynamic set** if it allows the operations **add**, **remove** and **contains**. The classic dynamic set data structures are **binary search trees** and **hash tables**, each with countless minor variations.
- In the **Java Collection Framework**, these are available as classes `TreeSet<E>` and `HashSet<E>`. Yes, you are still allowed to freely use all these collection classes even if you don't know how they work internally, any more than you know how a compiler or a graphics rendering engine works internally. That's what they are there for!
- Hash tables provide dynamic set operations in O(1) average time regardless of the set size *n*, although these could theoretically degenerate to linear O(*n*) operations no better than using an unsorted array. However, the probability of this happening is so vanishingly small as to be utterly meaningless in practice.
- (This linear time behaviour can always be artificially triggered by carefully choosing the elements and the order in which they are added into the table. Hash tables in applications that have to be resistant to attacks need to use good pseudorandom numbers as part of their hashing scheme of how they organize the elements internally.)
- **Balanced binary search trees** guarantee O(log *n*) operations no matter what, and allow efficient order-based operations **minimum**, **maximum**, **successor**, **predecessor**, **floor** and **ceiling** that cannot possibly be implemented efficiently for hash tables. Most of the time we have no use for such operations, making the hash table the best solution.

Lecture 10: Recursion

10.1. Self-similarity

- We have already seen that many problems can be broken down into smaller parts, and many of these parts occur so often in practice that there already exist good library methods for them. We can just call those methods to solve those parts, making our work easier since we only have to combine and solve the remaining parts of the problem.
- A thing is said to be **self-similar** if it contains a strictly smaller version of itself inside it as a proper part. (In mathematics, a thing is defined to be **infinite** if it contains an **isomorphic** copy of itself as its proper smaller part.)
- **Recursion**, a **method calling itself** for smaller parameter values, is often a natural way to solve self-similar problems after the self-similarity has been spotted.
- Spotting the self-similarity allows us to write **recursive definitions** for things we wish to compute. For example, the non-recursive definition for **factorial** $n!$ = 1 * 2 * … * (n-1) * n can be converted into a recursive definition $n!$ = (n - 1)$!$ * n.

- Recursive definitions can be a surprisingly powerful way to solve complex problems. The more complex and powerful the function $F(n)$ is working against you when you try to solve it, the harder it also works for you if you are able to place it to the right hand side of the definition $F(n) = s(F(n - 1))$ where s is some simple function (both sides of the equation balance by definition).
- "*Recursion is the root of computation since it trades description for time.*" (Alan Perlis)

10.2. Simplify and conquer

- To solve a recursively defined problem, first solve a smaller **subproblem**, a smaller instance of that very same problem, by having the method call itself. Then use the result of that subproblem to solve the original problem in a much easier way than computing that problem from scratch would have been.
- The subproblem is in turn solved by first solving an even smaller subproblem, which is in turn solved by first solving an even smaller subproblem, and so on.
- To avoid **infinite regress**, every recursion must have at least one **base case**, when the subproblem is so simple that you can just solve it on the spot without any more recursive calls. **Every recursive method must always start by testing for the base case**.
- A recursive function can have more than one base case, and could even have infinitely many base cases, especially if the recursive function takes two or more parameters.
- In theory of computation, both **recursion** and **iteration** are computationally universal, so any problem that can be solved one way, could always somehow also be solved the other way. However, some types of problems are better suited for recursion than iteration.
- The correctness of a recursive method can be proved using the mathematical **proof by induction**: show that the method works correctly for the base cases, and that if we assume that the method works correctly for all arguments less than n, then it also works correctly for the argument n. QED.
- Establishing the previous implication proves that the method works for all n. This truth climbs up the ladder all by itself, you don't need to do anything else.

10.3. Branching recursions

- True power of recursion is unleashed with **divide and conquer**, the recursive method making multiple recursive calls to **branch** the execution to multiple directions one at the time, or these days even in parallel with **multicore** processors.
- **Quicksort** and **mergesort** are classic examples of the power of divide and conquer, and these days **multicore parallelization**. Various **search algorithms** with multiple possible choices for each step are another.
- **Fractals** are aesthetic and natural geometric shapes that are self-similar and have a non-integer dimensionality. Self-similar **subdivision fractals** such as the **Sierpinski triangle** or **Quadric cross** that consist of smaller copies of themselves become natural and quite easy to compute and render with recursion.

- In mathematics, every fractal is infinitely deep, but when rendering it into a finite pixel raster, the base case of recursion is when the subshape becomes “small enough”, for example being smaller than the area of a single pixel.
- To trace the execution of a recursive method to better understand what is actually happening inside an iterative physical machine, you can have your method print out a debugging message with the current parameter values at the method enter and exit.
- The data passed back and forth the recursion should be in parameters and return values. **Avoid using data fields that are shared by all levels of recursion**. (Such fields need to be reset between calls, and even then this approach doesn’t work at all if the recursive method is executed **concurrently** by two or more **threads**.)

10.4. Downsides of recursion

- Since the strengths of recursion don’t really show up until third year or thereabouts in a decent CS undergrad program, among those who **program by superstition**, recursion tends to have a reputation of being inefficient and bad, never to be used in real programs. But few such people seem to be able to coherently explain why.
- That said, recursion really does have three downsides to be understood and avoided.
- **First**, every method call, recursive or not, creates a new **stack frame** of local variables and its **return address** of that method on top of the **stack**. A deep recursion simulating a loop (for example, through the indices of a potentially large array) can thus easily cause a **stack overflow**. The fixed size stack space reserved for each process is usually pretty small, no matter how many terabytes of virtual memory you have.
- **Second**, even without a stack overflow, the bookkeeping needed to maintain and update the stack makes the recursive method a constant factor slower than the equivalent iterative solution.
- **Third**, a branching recursive method making multiple calls can cause an **exponential chain reaction** of method calls. Sometimes this is inherent in problems whose search space truly is exponentially large, such as **Towers of Hanoi** or **Subset Sum**. However, this is horrendously inefficient when the same subproblems are computed over and over from scratch an exponential number of times.
- Easiest way to get around this tarpit would be **memoization**: use some auxiliary data structure to remember what subproblems you have already solved, and when the recursion comes to those subproblems again, just look up the previously **cached** result to return it in O(1) time.
- This is similar to solving a problem with **a lookup table**, except that the table is filled dynamically as its elements are encountered, as opposed to being hardcoded into the program at the source code level.

10.5. Some special techniques

- To prevent stack overflow, real programs that use recursion should always use a **recursion depth parameter** as a failsafe to cut off recursion once the limit has been reached, to prevent the misbehaving program from crashing in the end user’s machine.

- When there are multiple ways to subdivide a problem into smaller pieces, a more balanced subdivision produces fewer levels of recursion, O(*log n*) instead of O(*n*).
- A recursive call is a **tail call** if it is the last thing the method does, except for possibly returning the result up the line without modifying it.
- When any decent compiler recognizes a tail call, it can generate the low level instructions to simply reuse the current stack frame (whose contents will never be used for anything again) instead of generating the code to allocate a new **stack frame**.
- Recursion that consists of only tail calls is therefore completely immune to stack overflow even for humongous *n*.
- A handy technique to turn a recursive call into a tail call is to carry the current result of the computation along as an additional **accumulator parameter** of the recursive method. This trick can move all the computations after the call that prevent the call from being a tail call to take place in the calculation of this parameter and thus take place before the call, satisfying the letter of the law for tail calls.
- The further away the recursive call is from the tail end of the method, the more difficult it becomes to move all the code following the call into the calculation of the accumulator.