CCPS 209 Computer Science II Notes

This document lists the things taught in each lecture of the Ryerson Chang School course **CCPS 209 Computer Science II**, as taught by Ilkka Kokkarinen. It was originally compiled over two weeks back in January 2016 based on my experiences and memories from the many years that I had been teaching this course and whatever more or less witty things I managed to say in each lecture, and has been occasionally edited and extended ever since. We surely won't have time to explicitly cover every single item in this document, but I will try to explicitly point out everything that is important.

On March 28, 2020, the numbering of these chapters was edited to match the numbering used in the author's YouTube lecture video playlist "CCPS 209 Computer Science II".

Lecture 2: Classes and Objects

2.1. Classes and objects

- Back in the old times before the ascent of object-oriented programming, **imperative programming languages** such as **Fortran, C and Pascal** allowed the programmers to write **functions** that operate on passive data laid out in the memory. For example, `slice(bread);`
- In a more modern, and in many ways far superior **object oriented programming** approach, an **object** is an entity that combines together both the data and functionality that operates on that data. The object contains its own functionality, so now it is `bread.slice();`
- Or, depending on the way concepts are divided into classes, this might alternatively be `slicer.slice(bread);`
- Each object contains some **capabilities** (**methods**) that the outside world can **invoke** (**call**) to ask the object to perform that particular action. The functionality has been baked into the "live" data instead of being a separate, qualitatively different thing outside it.
- Objects are intended to model the entities of the **problem domain** that the program is written to perform useful computations on. For example, a bank account object might recognize the methods "deposit" and "withdraw", but not recognize the methods "fly", "get length" or "honk", since in the problem domain of banking, those operations would make no sense.
- An object also contains internal **data fields** to remember things that it needs during its lifetime. For example, a bank account object would obviously have to remember its `balance` and `owner`. These data fields would then greatly affect the behaviour of `withdraw` and other methods.
- 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.
- A new object is created from the class using the Java operator `new`, passing the constructor arguments to the operator `new`. The object continues to exist in memory as long as it remains

reachable from the live variables of your program, after which it gets automatically **garbage collected** after some unspecified time by the JVM background garbage collector.

- An object, **once created, cannot change its type or its location in the heap memory**. As long as that object exists, it will remain an instance of the very class that it was originally constructed from. The values of data fields can change during the object lifetime, but the type and the corresponding structure never will.
- 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**. (There is still a design choice with **transitive** verbs of whether they should belong to the linguistic subject or object. When a boy throws a ball, is "throw" written as a method in the class `Boy` or in class `Ball`?)
- Some objects constructed from the classes of the **Swing framework** can have a visual presence on the user screen independent of the **IDE** that is being used, and can listen to and react to the user actions done with a keyboard or mouse.

2.2. Instance fields to represent and store properties

- Variables declared inside the class are called **fields** (or **instance variables**) so that each object constructed from the class contains a separate copy of that field.
- The positioning of that variable declaration alone inside the class but outside any methods makes that variable to be a data field. No additional keywords whatsoever are needed to tell the compiler that some variable is a data field.
- All objects reside in memory area called **heap** so that, unlike the **local variables** of some methods that are stored in the **stack frame** in the **local variable stack**, the objects and their data fields persist in the heap as long as the object itself exists in the heap, regardless of the method calls and returns done by the method.
- Unlike local variables, instance fields have an **access modifier** `public` or `private` to control their visibility to the outside world. As a general rule, all data fields should always be `private`.
- (Leaving the access modifier out entirely denotes **package access** so that the member is `public` inside the same package, but `private` from everywhere else. This allows classes in the same package and under control of the same entity to have closer access to each other's internals than what is given to the outside world.)
- If some outside access to a private member is desired, you should instead 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.
- Good **object oriented design** exposes only the **semantic functionality** that objects implement to satisfy the needs of the outside world, and hides the implementation details of how the objects internally implement their behaviour. Data is **always** an internal implementation detail, and should therefore always be `private`.
- Objects can have arbitrary **properties**, some of which are implemented as fields, while some other properties are **virtual**, meaning that they are **computed on the fly** from other properties at the time that they are requested. In a properly designed class, the outside should in principle not be able to tell which properties are virtual inside the black box object.

- The Java naming convention for the mutator method to set the value of some property `Foo` is `setFoo`. The accessor method for that property is conventionally named `getFoo` as expected. (If that property happens to be a `boolean` truth value, the variant `isFoo` is also allowed.)
- Sometimes some piece of data is inherently something in the problem domain that is always the same for every object of that class. It would be redundant (and quite inefficient and error-prone, should the value of that common data change during execution) to store the same value separately inside every object constructed from that class. Making that field to be `static` causes one and the same copy to exist in memory, simultaneously shared by all objects and methods of this class.
- In retrospect, `shared` would have been a much better keyword for this purpose. The keyword `static` is used for historical reasons (in the programming lingo, "static" means compile time and "dynamic" means runtime). Too late to change the Java keywords and syntax now, though.

2.3. More on instance fields, static fields, and local variables

- 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. Local variables reside inside a **stack frame** created on top of **stack** at every method call. This stack frame is removed by the JVM when the execution returns from the method, making the lifetime of local variables to be that method only.
- If the same method is called again, its local variables are created from scratch inside the new stack frame, so they won't remember their values from their past lives.
- **Method parameters** are also local variables stored inside the stack frame along with the proper local variables, except that their initial values are provided by whoever calls the method. The method writer should never try to dictate them; ignoring the argument value given by the caller means that the parameter should have been a local variable in the first place.
- **Local variables do not have any access specifier**, since having one would not make any sense to begin with. Local variables don't even exist unless the method is being executed, so the issue of accessing a local variable when the control is outside the method cannot possibly arise.
- Variables that are declared `final` can never again be reassigned after their initialization. Using this keyword prevents certain types of bugs where you or somebody else modifies something that should not be modified.
- Note that the concepts `final` and `static` are independent of each other, so that any field could be both, either one, or neither. All four combinations are legal and meaningful.
- A field that is simultaneously `static` and `final` is called a **named constant**. There are many advantages to using named constants instead of writing **magic numbers** in your code. In addition to improved readability, changing the value of a named constant requires the change in one place and recompilation, instead of having to hunt all the occurrences of that constant in the code and edit them all, probably missing one or two and thus causing weird bugs in your code.
- Parameters and local variables can also be declared `final`, although this is merely a stylistic guideline that cannot affect the behaviour of other methods. Such extra verbosity does prevent some types of silly bugs in that method, though.
- Declaring some item of data to be `static` makes an irrevocable commitment that there can be exactly one piece of that data during the program execution. This might be the easy way to do

something quick and dirty, but also closes many doors from future improvements. For the same reason, programmers ought to eschew **global variables**.

- More generally, an important purpose of proper software engineering is to prevent the accumulation of **technical debt** that accumulates with **quick and dirty solutions** that may save time at the present moment but will impose exponential costs for the inevitable future redesign to prevent the system made up of "duct tape and bubblegum" from collapsing.

2.4. Constructors

- For reasons of both **safety** and **security**, the Java Virtual Machine that executes the compiled bytecode guarantees that every time any new object is allocated somewhere in the object heap memory, before anything else can happen to that object, those memory bytes are first filled with zeros to erase every possible remnant of the objects that previously resided in those bytes.
- This policy also conveniently guarantees that every primitive numerical field starts out its life initialized to value zero, and that all `boolean` fields start out their lives as being `false`. This makes Java **portable** between different computer architectures by guaranteeing that every Java program runs the same way every time in every computer that runs a standard-compliant JVM. This would not be the case at all if the object fields were initialized to whatever arbitrary values happened to be stored in those particular memory bytes at the time of object allocation.
- **Local variables** inside a method are not similarly initialized to zero values, but the Java language requires that every local variable must be explicitly initialized to some value before it is used.
- Occasionally you might want some fields to be initialized to some non-zero values. If you know that value at compile time, you can assign it to the field at its declaration. Otherwise, usually because the value of the field will be provided by the entity who needs the new object, you need to write a **constructor** to the class to allow initialization of the fields to values determined at runtime.
- A constructor is a special method that you never invoke explicitly. Instead, the JVM invokes it automatically every time a new object that is created. Inside the constructor, you will then write all statements that you want to happen to initialize that object and make it ready for action.
- Java recognizes your method to be a constructor from the rather silly syntactic rule that the **constructor has no return type,** not even `void`, and its **name must be exactly the same** (including the capitalization) as the class itself. (Some more modern languages have a separate keyword `constructor` or similar to make it clear what is what, instead of pursuing the false economy of trying to minimize the number of keywords in the language.)
- **Default constructor** is a constructor that doesn't take any parameters. If you don't write any constructors into your class, the compiler automatically synthesizes a do-nothing default constructor, so that objects can be created. But as soon as you explicitly define even one constructor, this synthesization of default constructor no longer takes place, and new objects can be created using only the constructors that you have explicitly provided.
- You should prefer using constructors rather than **initializers** for the fields, unless the field is initialized with a constant literal. Since the field initializers are executed in the order that these fields are defined inside the class source code, initializing a field with a method call can cause this method to be executed before the rest of the fields have been initialized (and are therefore

still zeroes due to the above guarantee of object memory allocation). This imposes a silent **order dependency** on the fields defined inside the class that may cause the class to silently break when somebody rearranges its members in the future.

2.5. Primitive types

- The Java language makes a fundamental distinction between its eight **primitive types** (`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`) that are defined and set in stone in the language itself, versus all **classes**, any number of which can be freely written in the future.
- These eight are the only primitive types that will ever exist in Java. They are, in a sense, the "elementary particles" that all higher level data consists of, analogous to the way that molecules are built up from atoms from a small fixed set of elements. (Or how arbitrary complex and long proteins and other biological structures and superstructures are built from a small number of different amino acids. Or how words, sentences, paragraphs and arbitrary complex novels up to "War and Peace" can be built up from a small fixed set of possible letters and other characters.)
- In fact, analogous to how the physical entities that were once erroneously thought as indivisible "atoms" were later discovered to be divisible to even smaller parts, our primitive types are also not the actual elementary particles of computing. They are all composed of **bytes**, which are in turn composed of **bits** in memory, each one being either 0 or 1. One can't really get any simpler than that. These bits and their operations are what all computing is made of, since in principle all computation is merely **syntactic sugar** for **propositional logic**.
- No new primitive types can be created on top of these eight. Even when its element type is some primitive type, the corresponding **array** type of any dimensionality is a class.
- The primitive types correspond to the data types that the processor machine code can directly and natively operate on, so that the operations on these types translate directly to the machine code instructions for efficiency.
- To guarantee portability between different architectures, the Java language **guarantees the sizes, bitwise encoding and potential overflow behaviour of all four primitive integer types**.
- The same guarantee is **not** in effect for **floating point arithmetic**, although for those extremely rare cases that such portability is needed, the use of **IEEE 754 floating point standard** can be enforced with the keyword `strictfp` applied to either one variable, method or the entire class.
- Like every serious system designed over the past couple of decades, Java is fully standardized to use **Unicode character encoding** in the `char` and `String` types to guarantee that all **textual data** is fully portable between systems and never unintentionally corrupted in computations.
- The Java standard library offers a **wrapper class** for each primitive type, for example `Integer` for `int`, to allow a primitive value to be wrapped inside an object. **Autoboxing** can convert automatically between the primitive type and the corresponding wrapper as needed, with a few caveats explained later, mostly related to the notion of object equality versus value equality.
- These wrapper classes also offer a bunch of useful `static` methods for common operations for that particular type, especially the methods to determine the Unicode properties of arbitrary characters implemented as `static` methods in `Character`.

2.6. Primitive types versus references

- A variable whose type is one of the eight primitive types **contains the stored value inside the variable itself**, in the very bytes that comprise the variable itself. (A **variable**, in computer science, is defined as "a named memory location that contains a value". Those bytes will continue to exist in memory separately of how our code assigns names to them according to the **scope** and **lifetime** of those variables.)
- However, every variable whose type is any class (including `String`, which could otherwise *almost* be treated as if it were the ninth primitive type) is a **reference** that contains the **memory address** of the object that is stored in heap.
- This distinction of objects and the variables that refer to them has many important consequences that would baffle anyone who incorrectly assume that each variable contains the object itself the same way as the primitive types do, instead of really containing the directions of how to find that object using its memory address.
- Primitive types take different numbers of bytes to store. However, since every memory address is the same, regardless of what sort of value the bytes there represent, every reference in Java takes the same number of bytes of memory to store and represent.
- For primitive types, the **equality comparison** operator == compares their values. For references, the operator == still compares their values, but those values happen to be memory addresses, not the object contents. **The result of comparison of references using == can never depend on the object contents,** because this operator checks only whether both objects are the exact same object in the same memory address.
- The memory address gives every object **a unique identity** that separates it from all other live objects. Even if two objects were 100% identical clones of each other in every possible way with respect to their types and values of data fields, the fact that they reside in two separate memory addresses allows us to tell these two objects apart.
- After some object has become unreachable from your code once you lose all references to it, and the **garbage collection** has made that part of the heap memory for future allocations, another object might well be created in the same memory address. However, your program won't be able to tell even in principle that this has happened, since your program no longer has any live references to that memory location that it could compare its new reference to!
- Garbage collection allows programs to run for an arbitrary long time without worrying about running out of memory. In more primitive languages where the memory must be released explicitly back to the system, the twin dangers of Scylla and Charybdis of **memory leaks** and **dangling references** cause all kinds of difficult and unpredictable bugs in programs.
- A Java program can still have an indirect memory leak in that some reference that you will never again use keeps the object alive. To prevent this, set a reference to be `null` once you know that you will never again use the object that it points to.
- Since all process memory is released when the process terminates, programs whose runs are short could theoretically run without any garbage collection. **Embedded systems** are usually meant to run uninterrupted for a long time, even for years or decades, and therefore should not leak any memory over time, not even an almost unnoticeably slow drip.

- (On the other hand, there is the tale of a guidance system inside a missile, where the designers simply removed the garbage collection altogether and put in twice as much physical RAM as is needed to cover the theoretical maximum of memory allocations during the maximum flight time of that missile. After the execution of the guidance system program, there was no longer need for any garbage collection. At least not within the program that no longer seeks to conquer the human, nor to be conquered by the human. This is all totally "warrior Zen", or Tao, or something else vaguely exotic of that nature as long as it makes me look cultured and smart.)
- Java does not have **pointer arithmetic** that would allow the programmer to treat a memory address stored inside a reference as an integer (that it really is, in the bytewise level) and sneakily store it in some integer variable that way, and therefore does not have any of the myriad safety or security problems normally associated with pointers in languages such as C.
- If you want to compare two objects for **content equality**, you should always use the method `equals`, calling this method for one object and passing the other one as an argument. Since any well-defined equality relation is **symmetric**, it should never matter whether you make this call in the order `a.equals(b)` or `b.equals(a)`. (Although see below.)

2.7. References as method arguments

- When executing some method call of the form `obj.method(args)`, the JVM follows the reference `obj` to the actual object residing in the heap, and calls its `method` with the given `args`.
- Two references are said to **alias** if they contain the memory address to one and the same object. Such aliasing happens most commonly when you call a method whose parameter type is a class instead of a primitive type.
- All parameter passing in Java is always done **by value**. However, passing a reference to a method by value causes the caller and the method to share the same object. If the method execution modifies the internal state of that object, that modification persists for the time that the method terminates and the control returns to the caller.
- Passing a reference to an object by value can be used to simulate passing the referred object to a method by reference. The method can modify the internal state of the object, but cannot modify the original reference to the object.
- This phenomenon is most commonly seen with **methods that operate on arrays** that can contain (even hundreds of) millions of elements. It would be highly inefficient to create a separate identical copy of that array object to be passed to the method at every method call. Instead, the method receives only the memory address where the entire array object resides. It therefore takes the exact same amount of execution time to pass a million element array to a method as it would take to pass a tiny ten element array.
- This is most commonly seen in recursions that operate on arrays, since it would be inefficient to extract the particular subarray under interest into a separate array object. Instead, the entire array is passed by reference to each level of recursion, and additional **index parameters** (typically `start` and `end`, and sometimes the single parameter `n` to denote using the first `n` elements) tell the method which particular subarray it is supposed to only operate in.
- References can also have a special value `null` that cannot exist for primitive types, because all their possible bit value combinations are already in use to mean something else. Inside the JVM,

`null` is encoded as bytes that are all zeros **to indicate that the reference is not really pointing to any object at the time.** All reference fields are guaranteed to become `null` at object creation.

- Any method call made through a reference whose value is `null` is guaranteed to crash the method at runtime. (So actually, it *does* matter whether you say `a.equals(b)` or `b.equals(a)` to compare two objects for content equality, in a situation where either reference `a` or `b` could potentially be `null`.)
- The `null` references are the infamous "billion dollar mistake" of software engineering. Modern languages such as Swift incorporate into language hard compile time guarantees that some reference cannot be `null`, so that the method receiving such a reference does not need to go through the rigmarole of checking for `null` at runtime and can still safely use that reference.
- Try as you might, the lack of unsafe **pointer arithmetic** in Java makes it impossible to produce a reference that points to some memory location that does not currently contain a live object in the object heap. Throughout its entire lifetime, every reference is either `null`, or is assigned a value from some other expression that points to a live object in the heap. Programming in Java is guaranteed to obey **memory safety**.
- This maintains the global **invariant** of every reference always being legal in this sense, since logically there cannot exist the first time that an illegal reference gets initialized!

2.8. Immutable versus final

- Same way as with variables whose type was one of the eight primitives, a reference can be declared to be `final` so that it cannot be later reassigned to point to some other object.
- Because a reference variable is a separate entity from the object that it points to, the keyword `final` affects only that reference. The same object can be simultaneously pointed from multiple references, some of which are `final` while some others are not. Therefore being `final` cannot possibly be a property of that object itself, but of the name that refers to the said object.
- Especially declaring a reference to be `final` does not prevent the method calls through that reference from modifying the internal state of the object.
- For this reason, `final` should not be confused with **immutable**, a related concept that means that the class has been intentionally designed so that **it has no public mutator methods**, but accessors only. An object constructed from an immutable type cannot change its internal state later during execution, at least not in any way that could be detected from outside by calling the `public` methods of that object.
- (Objects that are immutable from the point of view of the outside world can still optimize their execution time by internally **caching** the results of some computations that they might again need later, that way modifying their internal state. However, this does not affect the results that any methods of that **black box** object return to the outside callers, except that those same results will be returned faster.)
- Immutability provides a whole bunch of surprising advantages in programming, which is why it is recommended to design your classes to be immutable whenever possible and convenient. However, immutability has also one giant downside in situations where the code needs to iterate through a large set of different values and would therefore require the creation of a separate object for each such value, which is why we don't automatically make all data immutable.

- For example, `String` is intentionally defined to be an immutable type in Java. To allow us to sidestep the previous downsides, `StringBuilder` is offered as the mutable version of that class, optimized for the common operation of `append`ing more characters into the end of the string by keeping extra slack space in the underlying character array. (In computing, we can often **trade memory for time and vice versa**. Since these days we have more memory available than we know how to use it all, it is usually a good trade to make your program run faster.)
- Building a long answer string piecemeal using immutable strings would require an O(n) copy of the entire string for every single individual character `append`, making the result building to be at least O(n^2). Since the `StringBuilder` offers an O(1) amortized time operation `append`, the result can be built up in only O(n) time.

2.9. The implicit reference `this`

- Sometimes it would be beneficial for some instance method to be able to find out which particular object it is currently being executed on. For this purpose, every instance method automatically contains a special local variable `this` that does not need to be declared and that points to that very object. (Python uses the conventional name `self` for this **implicit parameter** of each instance method.)
- Most of the time using `this` is redundant, because when some method call `foo()` is written without the explicit object reference prefix, the compiler expands this call as if you had written `this.foo()` out in full to call the method `foo` for the same object `this` that the current method is being executed on.
- Sometimes using `this` is unavoidable and necessary when the method has to call another method that needs to know the identity of the original object as one of its arguments. (This situation occurs in object-oriented design most often with the so-called **callback** methods where the computation calls the method in a given object once it has reached its goal.)
- When you give the constructor parameter the exact same name as the field that that parameter will be assigned to, the parameter **shadows** the field with the same name in the statements of the method body. The reference `this` can then be used to force the compiler to treat that variable name as a field in the left hand side of the assignment.
- Back in the day, the reference `this` was merely a morsel of syntactic sugar to turn a non-object oriented language into an object-oriented one, albeit without inheritance and polymorphism. A method call `obj.foo(args)` can be converted by the compiler into an ordinary imperative programming function call `foo(obj, args)`, with the implicit parameter `this` receiving its value from the extra first parameter that refers to the implicit object for which the method call is being made.
- Methods that are declared `static` are called without the implicit parameter `this`. Instead of prefixing the method call with an object reference, the call is instead prefixed with the name of the class that contains the definition of the `static` method, such as `Math.sin(2)` or `Character.isWhitespace(ch)`, to resolve any ambiguity in the potential situation where multiple classes define a `static` method with an identical signature.

- Same way as the reference `this` can be left out when calling an instance method in the same class, the class name prefix can be left out when calling a `static` method that is defined in the same class.
- A method should be written to be `static` if it does not need anything from the underlying object, but has everything that it ever needs to compute its result in its parameters and the `static` fields of the class. A non-static method can always be later turned `static`, if need be.
- A class that contains nothing but `static` methods for common operations for some particular data type and is not designed for instance creation is called a **utility class** for that data type. The Java naming convention is to name such utility classes after the data type but pluralized, such as `Arrays` or `Collections`.

2.10. Method overloading

- Unlike in our natural language, the word **overloading** has no negative connotations in object oriented programming. In this context, this technical term simply means that the class defines several methods with the exact same name but different parameter lists.
- Using the exact same name for all these methods immediately conveys to the reader that the methods are **semantically identical**. These methods achieve the same purpose even though they happen to take and operate on different types of arguments.
- The purpose of all names is to instantly convey some useful information to the human reader to help the programmer communicate the purpose of his code to other programmers. The language and its compiler don't care about the chosen names, and couldn't care less even if you named your methods `method1`, `method2`, `method3`, ...
- The overloaded methods could just as well have been given different names, but in general it is bad style (and a **code smell**) to have your method name talk about the data types that it receives as parameters.
- **The parameter lists of overloaded methods have to be different** even ignoring the parameter names so that the compiler can always determine without ambiguity which method is now being called. The overloaded versions may each have a different access modifier and return type, but these have no effect on method overloading resolution.
- Having two overloaded methods taking parameters of different primitive types is perfectly legal, but note that adding a second method `foo(int x)` to the class that already has the method `foo(double x)` causes the call `foo(42)` to no longer be bound to the second method.
- It is legal for a class to have two overloaded methods that differ only by the order of their types of their parameters. But this still does not mean that you should ever do this, since then everyone has to remember each time which one of your two such methods was which, this mental effort no longer available for other, far more important things. You are encumbering the proverbial camel with an extra straw for zero benefit, and eventually such little straws, each straw by itself as light as to be almost meaningless, will add up to break that camel's back.
- Constructors can be overloaded the exact same way as with ordinary methods. The choice of which constructor gets executed is based on the types of the arguments given to `new`.
- Sometimes overloaded constructors contain duplicated code, which is always a code smell. Java does have an odd-looking special syntax to call a constructor from another constructor to

eliminate such duplication, by using `this` as a method name. However, such calls are further restricted so that such a call must always be the first statement in the constructor.

- Overloading should not be confused with **method overriding**, a very different but an extremely important concept for the rest of this course when we talk about inheritance and polymorphism. We could easily program all of our complex systems without method overloading and not lose anything important. But without method overriding, everything in Java would collapse and we might as well put on our brown seventies pants and go back to Basic or C.

2.11. Enumerated types

- Once a class has been defined and compiled, any number of new objects could be constructed from it. But sometimes the class corresponds to some concept in the problem domain that is inherently limited so that the set of its possible objects is fully known and set in stone at compile time, and it would be a semantic error for additional objects of that type to ever exist.
- For example, consider a class whose objects represent the seven days of the week. There should exist exactly seven objects for these seven days, and never more.
- To define a class that represents this kind of concept, use the keyword `enum` instead of `class` in its definition to define it as an **enumerated type**. Enumerated types are classes for which the `new` operator has been disabled so that trying to use it is always a compile time error.
- The old way before Java 5 to implement this sort of a class was to make its constructor `private` to ensure that no new objects can be created from the outside code, and then define a `public static final` named constant inside the class body for each legal value of this type to refer to the object that represents that particular value.
- An even older way in languages before object oriented programming was to use `int` as this enumerated type, and then simply define that 0 stands for the first particular value, 1 stands for the second one, and so on. Encoding enumerations as integers provides zero type safety since any `int` variable can be assigned any integer value regardless of our noble intentions that only the first *n* natural numbers should be used.
- Furthermore, looking at an `int` variable we cannot tell whether it is semantically meant to be an actual integer or to encode an enumerated type. Using multiple enumeration types in the same program also allows arbitrary "cross pollination" between these types to give birth to mysterious bugs that the compiler cannot prevent.
- Many methods in the Java standard library still are relics of that kind of thinking in that their **option constants** are given as integers that are declared as named constants inside that class. Had these enumerated types been part of the Java language since day one, these option constants would have surely been defined as enumerated types.
- In the beginning of the `enum` body, you must list the names of the objects to be automatically created by the JVM when the class bytecode is loaded in memory. This is actually just syntactic sugar for those `public static final` named constants that the compiler turns these freely floating names into when it is compiling an `enum`.
- After all that, the enumerated type is still real class and can therefore define arbitrary `static` and instance fields and methods, even constructors. The constructor arguments are given in parentheses after the name of each individual value.

- Inside every enumerated type, the compiler will also automatically generate a useful bunch of additional instance and class methods such as `ordinal()` and `values()` that allow these enumeration objects to be converted to and from integers for the purposes of old-timey operations, and iterated one at the time using a for-each loop.
- Enumerated types are usually defined to be immutable, but this is not any law of nature or man.
- Be careful in making assumptions about some problem domain concept being inherently set in stone, unless it really is that way by definition. (And even definitions that all of humanity agrees on today might still change in the future, although probably not the agreement that one week consists of exactly seven days.)

Lecture 3: Inheritance and Polymorphism

3.1. Types in programming languages

- During the execution of a program, some **data objects** exist in the memory for the code to operate on. Even though all these objects are really nothing but raw bytes, we prefer to impose a convenient fiction of **higher level semantics** upon them to make them easier to reason about.
- Any higher level entities defined in the language are fictional the same way that, say, hobbits and vampires are fictional. Despite that, these fictional entities such as "functions" or "loops" can still be used as convenient shorthands in expressing ideas that are real, at least real in the sense that their consequences are real as far as our behaviour is concerned.
- In the **type system**, a **type** defines all those properties that remain constant during the entire lifetime of each object constructed from that type.
- The purpose of types is to prevent **type errors** in programming by setting up voluntary but binding constraints of what is possible and legal to do with given data. In Java and other **explicitly typed** programming languages, these type errors are revealed at compile time.
- It is always better to find errors as soon as possible so that you know to turn back whenever you are heading towards inevitable failure.
- Java is a rather heavy and verbose language for programmers, designed to reveal errors at compile time. Contrast this to Python, Ruby and similar languages that instead optimize for ease and brevity, leaving most of the error detection above basic syntax rules for **unit testing**.
- One comment to the article "If programming languages were weapons..." that made rounds on the Internet a few years ago gave Java a much more accurate description than the original article, calling it "a belt fed 240G automatic weapon that tries hard to avoid gun accidents: each round has to be individually authorized by ticking off a detailed questionnaire. By the time you get to actually shoot, someone may have already beaten you to death with a stick."
- In Java and all similar object-oriented programming languages, a new type can be defined by writing a **class**. From this one and the same class, any number of separate **instances** can then be created. In object-oriented programming, these instances are called **objects**. Each object contains the **fields** and **methods** defined in the class, and **constructors** to initialize the fields to the desired non-zero values.

- These different instances of the same type are **structurally identical**, but store different values in their data fields and reside in different memory addresses, which allows us to tell them apart even if their data field contents happen to be identical.
- In **object oriented design**, each class would ideally correspond to one **concept** in the **problem domain**. If the problem domain consists of *n* different concepts, the program would consist of *n* classes, with the **verbs** of the concepts becoming the methods of these classes. Difficulties in thinking up short descriptive names for classes and methods tend to strongly suggest bad design choices in choosing these classes and methods.
- The problem level concepts tend to form hierarchies so that both chequing accounts and savings accounts are bank accounts, and both hawks and sparrows are birds. Birds in turn are animals, and animals are objects.
- **Properties that can change over the lifetime of the object should not be distinguished using the type system**. For example, it would be quite reasonable for the types `ChequingAccount` and `SavingsAccount` to exist within the same program, but wholly unreasonable to have `BankAccountWith1000Dollars` and `BankAccountWith500Dollars` as separate types. Since the balance of the particular bank account can change over time, it should instead be implemented as a data field inside the class.
- **Types should be used to model variation in behaviour, never the dynamic variation of state.**
- If we only have classes to express types but no inheritance to organize them (as it were in languages that predate object oriented programming), it becomes impossible to express the notion that the problem domain concepts of `Hawk` and `Sparrow` are somehow "closer" to each other than, say, the concepts of `Hawk` and `Planet` are to each other. Every type would be an island, as far as the type system was concerned.

3.2. The need for inheritance to avoid repeated code

- Unlike in the physical world, inside the pure perfection of mathematics there should **never be any need to repeat anything**. For example, all people can simultaneously think of and use the same number 7, instead of each person needing to possess his or her own personal copy of that mathematical object. Since computer programs are mathematical objects, the same principle should naturally apply also to them.
- The **DRY Principle** (**Don't Repeat Yourself**) instructs us to **never say anything twice** in the same program. The related **Single Point Of Truth Principle** states that every **contingent truth** about the behaviour of the program should be stated **exactly in one place** inside the program.
- (Simplest application of the Single Point of Truth Principle is to require that you are not allowed to write any **magic numbers** other than 0 and 1 in your code, and all other constant literals must be given descriptive names as `static final` **named constants**. This makes it easier to understand what is going on, and to modify these named values later if need be.)
- Inheritance allows subclasses to inherit a method implementation instead of the programmer having to copy-paste the same method to all subclasses. However, contrary to the common misconception about object oriented programming, this is certainly a nice thing to have, but not even close to being the main benefit of the inheritance mechanism.

- Without inheritance and static type checking, it would not be possible to write in a type safe fashion a **polymorphic** method `void pluck(Bird b)` that could be given as argument any type of bird, but not any planet, bank account or car without triggering a compile time error. You would end up writing two essentially identical methods `void pluck(Hawk h)` and `void pluck(Sparrow s)` only to keep the type checking formalism happy.
- Even worse, every time some new subtype of bird was introduced, you would have to write **yet another** (essentially identical except for the parameter type) version of this same method, repeating yourself *ad nauseam*. We therefore firmly reject any such approach.
- Of course, you could completely toss out all **type safety** and write the method `pluck` to liberally accept any object whatsoever as argument, even though you intend to only ever call it with birds. But what would happen if you or somebody else called this method with, say, a `String` or a `BankAccount` object anyway, and the method then tried to call its method `fly` ? The only possible outcome would be a **runtime crash**, which would as per Murphy's law most likely occur at the worst possible moment during the deployment of your program.
- **Static type checking** exists in all such programming languages **to prevent runtime crashes** due to the program trying to do something logically impossible in the problem domain, such as asking a chequing account to fly, or trying to deposit a hundred dollars inside an owl.
- In Java, the static type checking is strong enough to guarantee that it is **impossible** to create a situation where the program crashes at runtime because it attempts to call some method that doesn't exist in that object. (A Java program can still crash for various other reasons that are algorithmically impossible to detect in general during compilation.)
- **Dynamic languages such as Python or Ruby**, as they have no explicit type system at the source code level, cannot make a similar guarantee. This is the price we pay for the extra flexibility of the **duck typing** in line of the dynamic philosophy of those languages. (Decent use of **unit testing** alleviates this problem so that it is not really any problem in practice. If your Python program actually crashes at runtime due to typing error, that just merely shows that your unit testing has been woefully insufficient.)
- Contrary to the common misconception, **Python is still a strongly typed language**. Every object has a type that cannot change after the object creation. But unlike Java, Python typing is not **explicit** in the source code, although some extension modules allow the use of **type hints**.
- (Also contrary to another, even more commonly repeated misconception, **Python is compiled**. Yes, Virginia, it really is. This compilation, just like it has taken place in every serious interpreter in every "interpreted" language written over the past three decades, merely happens silently after the textual source code has been read in, instead of the programmer having to explicitly initiate this compilation process. This compiled bytecode is saved in `*.pyc` files for future use.)

3.3. Inheritance between classes

- A class can be defined to **extend** some **superclass**. This causes the subclass to **inherit** all the **fields** and **methods** of the superclass, almost (although not really, we will come back to this issue) as if they had been copy-pasted into the class body.

- Proper inheritance should always model an **is-a relationship** between the problem domain concepts that these two classes represent. Therefore it is sensible to say `class Car extends Vehicle`, but it would be utter nonsense to say `class Car extends Engine`.
- Every car **has** an engine, but the car itself **is not** an engine, since its **public interface of methods and resulting behaviour are completely different**. The public interface is a tool of **abstraction** that lists the **essential features** of the system that cannot be changed without changing the essence of the system, as opposed to its **accidental implementation details** that can be freely changed as long as the public interface does not change.
- The **has-a relationship** is better modeled with ordinary **composition** with object fields. This can be either one-to-one (such as a car and its engine) or one-to-many (such as a car and its wheels).
- Since a Java object cannot change its type after it has been constructed, **inheritance should only be used to express variation within the kind that remains constant during the lifetime of the object**, such as the species of the animal. Dynamic aspects such as age, weight, or the physical location of each individual animal are better modelled as object fields.
- Since an object can never change its type after construction, the class `Car` being a separate subtype of `Vehicle` requires that `Car` has no **mutator** methods that could turn some car object into an `Airplane` or a `Submarine` in the spirit of some 1970's movies of James Bond and similar spirit, but every car will forever solidly remain a car until it has been scrapped.
- In problem domains where higher level things built up from smaller parts can be reconfigured to create new things in the style and spirit of Legos, the high level classes represent these configurations as **compositions** of low level objects, and the objects constructed from those classes are supposed to exist only as long as that configuration exists. (In a sense, ordinary classes also do this, with the raw memory bytes being these Lego building blocks.)
- The individual Lego blocks are immutable, but the exact same individual pieces can be put together in many different ways to create either a `Spaceship` or a `Castle`. The individual pieces are exactly the same in both structures, but the general structure makes the difference in the higher level semantics of the composition of these pieces.
- Furthermore, somebody might use these Lego pieces to build up some entirely new kind of useful structure that we could not even imagine during the original design and our language therefore does not have a word for.
- In the Java type system, **the subclass objects are at the same time also superclass objects**. For example, every `Hawk` object is at the same time a `Bird` object. Note that `Hawk` and `Bird` are still separate classes in the type hierarchy: the same object simultaneously has two different types as needed by the rest of the program.
- Variables of type `Bird` can also refer to objects of type `Hawk`, or any other subtype of `Bird`. This is necessary for us to be able to write type-safe polymorphic methods that can take arbitrary birds as parameters that they operate on.
- The subclass can also have additional members that do not exist in the superclass. The subclass objects will then have these additional members, whereas the superclass objects will not.
- Through a reference of type `Bird`, you can only call methods that exist in the class `Bird`, even if you knew for a fact that the actual subclass object that the reference points to has that method. **All algorithmically decidable type systems that allow universal computation are necessarily conservative in what they accept.**

- If the subclass `Owl` defines a new method `hoot`, and you declare `Bird hedwig = new Owl();` the call `hedwig.hoot();` will then not compile since the variable `hedwig` is a `Bird`, and there is no method `hoot` in the class `Bird`. **The compiler can never make any decisions based on object types, since objects do not exist at the time of compilation!**

3.4. Overriding inherited methods

- The subclass definition can also **override** an inherited method simply by providing a new version of that method, with the **exact same name and parameter types**.
- **Only methods can ever be overridden, fields can not.** This is why you should never make fields `public`, but define **public accessor methods** to read and write them. This way future subclasses can change the way some attribute is defined inside the class, for example making it a **virtual attribute** whose value is computed on demand based on other attributes.
- Since it is very easy to accidentally merely **overload** a method instead of **overriding** it as intended, it is a good idea to always tack the `@Override` **annotation** in front of each overridden method. Such **syntactic salt** helps the compiler catch these easy mistakes at compile time. (Some modern languages even have `override` as a required keyword.)
- When overriding a method, its parameter list must be **exactly the same** as in the superclass version, The **return type can be made more specific** in the subclass version. For example, if the superclass method returns a `Bird`, the overridden method can return a `Hawk`. Since every `Hawk` is also a `Bird`, the caller of the subclass method gets what it is promised to get.
- For example, the class `Bird` could have a method `Egg layEgg()` that returns an `Egg`. This is the most that we can say in this superclass level. However, the subclass `Hawk` could override this method with the signature `HawkEgg layEgg()`, if the class `HawkEgg` is a subclass of `Egg`.
- The implementation of the overriding subclass method does not need to have anything in common with the superclass version of that method. The compiler would not be able to enforce that any semantic properties were maintained anyway. However, the famous **Liskov Substitution Principle** that we will introduce later will define precisely what constitutes proper overriding.
- If some method is declared `final`, the subclasses are not allowed to override it, but they have to take that method implementation exactly as it is defined in the superclass. If the entire class is declared `final`, no further subclasses can be extended from it. (For example, `String`.)
- This use of the keyword `final` should not be confused by declaring some variable to be `final`, meaning that the variable can't be reassigned after initialization. Nor should it be confused with designing the class to be **immutable**, meaning that an object constructed from that class cannot change after construction.
- When designing a class, you should document **whether the class itself uses some of its methods that subclasses can override**, that is, whether those methods are **template methods**. That way, the subclass writers cannot accidentally break the behaviour of superclass methods.
- For example, all other methods of `Random` are guaranteed to use the template method `next` internally to generate the random bits they need. When extending this class, it is sufficient to override only the `next` method.

3.5. Polymorphism and dynamic binding

- Since class members must be preserved in inheritance, **the subclass is guaranteed to have all the public members of the superclass**. For example, if the superclass `Bird` has a method `fly`, every subclass of that class is automatically guaranteed to also have a method `fly`, although this method may have been **overridden** to do something different in different subclasses.
- This allows us to write **parameter polymorphic methods**, that is, **methods whose parameter type is some superclass**. The one and the same method can then operate on **all current and future subtypes** of its parameter type, instead of us having to always write yet another redundant version of this same method whenever a new subtype is introduced.
- For example, the polymorphic method `void pluck(Bird b)`, written and compiled only once, can be passed as an argument an instance of `Hawk`, `Owl`, `Albatross`, … even if that particular subtype of `Bird` did not even exist at the time that the original method `foo` was compiled!
- Since the method `pluck` solidly depends on a high level abstraction `Bird` that does not change, it never needs to be recompiled no matter what weird new subtypes of `Bird` people will come to create. Polymorphic methods are **future compatible** with all the subtypes that anybody will ever create in the future, even when the method itself is written, compiled and set in stone only once!
- If the body of the polymorphic method `pluck` makes the call `tweety.fly()`, this call **cannot possibly be bound at compile time**, because this method call is supposed to do a different thing when called for different subtypes of birds.
- For this reason, in Java, method calls done for an object are **bound dynamically** at the time when the call is actually executed. The JVM follows the reference `tweety` to the object in the heap and looks at its internal **type id field** to choose on the fly (heh) the correct version of the method `fly` to be executed at that call.
- The very same method call will therefore do a different thing depending on the subtype of the object for which you call that method. This makes polymorphic methods possible.

3.6. Abstract superclasses and methods

- Sometimes the problem domain concepts that the superclass models are so **abstract** that in the superclass level, it is flat out **impossible** to give any meaningful concrete implementation for some particular method.
- For example, we can describe how a hawk flies, and on the other hand, how very differently from that a hummingbird flies. However, how does a bird fly? The concept of bird is simply too **abstract** for us to give an answer, even though every bird flies. (In our toy problem domain, penguins and ostriches don't exist. Every model must be a simplification of the problem domain in some way, otherwise you might as well use the problem domain directly as its own model.)
- However, if the method `fly` were not defined in the superclass `Bird`, it would be possible for somebody to create a subtype of `Bird` that cannot fly. The polymorphic methods that operate on birds then could not assume that the method `fly` exists.

- In this situation, the method `fly` should exist in the superclass `Bird` to guarantee that all **concrete** subclasses of `Bird` will have it, but be declared `abstract` in `Bird` and given no method body at all, not even an empty body.
- If the class has one `abstract` method, the entire class itself must also be declared `abstract`. This declaration **prohibits objects from being created from this class** with the operator `new`. (If such an object were allowed to exist, what would happen at runtime when somebody eventually created that object and called its `abstract` method?)
- **An abstract type is useful as the parameter type in a polymorphic method.** This method will then be passed as arguments objects from various concrete subclasses of `Bird`.
- **All four possible combinations abstract-abstract, abstract-concrete, concrete-abstract and concrete-concrete are legal and meaningful** when it comes to a superclass and its subclasses.

3.7. Visibility of class members

- An object created from subclass has all the fields and methods that were defined in its superclass, even those that are `private`. However, the access modifier determines whether these fields and methods are visible to the methods of this subclass.
- Members declared `private` can be accessed **inside the very same class only**. The subclass methods cannot see these fields and methods, even though they do exist in the object!
- The only way the subclass methods can indirectly access the `private` members of the superclass is by calling inherited methods that access them. Since the inherited methods are defined in the superclass, they can see and access the `private` superclass members.
- If you have overridden a superclass method that you would still like to invoke from a subclass method because it does something useful that the subclass version would need, use the keyword `super` as a “pseudo-object” in the method call to explicitly force the superclass version of that method to be executed.
- Members declared `public` can be accessed **from anywhere** that has a reference to the object that contains them. The inheritance relationship does not affect this in any way.
- Members declared `protected` can be accessed in the **same class and subclasses**, and from all the classes that are part of the same package.
- Data fields inside a class should always be declared `private`. If some field needs to be accessible from the subclass methods, provide a `protected` accessor method for that field.
- Requiring accessor methods to access fields that represent **attributes** also gives you leeway to later change that attribute to be **virtual**, that is, computed dynamically in the accessor method at the time of the call. For example, a `Length` class that stores lengths as centimeters can provide an accessor method for inches, and vice versa. (Or furlongs, if somebody needs this in the future.)

3.8. The universal superclass `Object`

- In the inheritance hierarchy of Java classes, the universal superclass `java.lang.Object` is a superclass of all other classes, either directly or indirectly. Starting from any class and climbing up the inheritance tree as high as you can, you will **always** end up in `Object`.
- If a class does not explicitly extend another class, it implicitly extends `Object`.

- `Object` itself has no superclass. Even more amazingly, the universal superclass `Object` is not `abstract`, although any actual need to create objects of type `Object` is now long obsolete.
- **The existence of `Object` type allows us to declare reference variables and parameters that can refer to any object whatsoever.** Of course, you can only call methods that exist in the universal superclass `Object` through such reference.
- The superclass `Object` defines a bunch of handy and by now surely familiar methods such as `toString`, `hashCode`, `clone`, `equals` and `finalize` that are therefore **guaranteed to exist by law in all classes** in Java.
- Most of these methods, especially `toString`, have a rather useless default implementation in `Object` that is best overridden in your subclasses.
- When overloading the method `toString`, it is also a good idea to provide a public accessor method for every field whose value appears in the generated string, so that users of this class won't need to parse that information from that string, making the user code dependent on that particular implementation of `toString`.
- The default implementation of `equals` simply checks whether the objects are the same object stored in the same memory address. This is the most that the method can do knowing that both objects are to be compared are some kind of objects. In general, the higher up you go in the inheritance hierarchy, the less you can assume and say about the actual objects.
- Since there are no guarantees of when the garbage collection will release an object, you should not rely only on the `finalize` method to release important resources held by an object, but release the resources explicitly once you know that you will no longer need them.
- Overriding `finalize` to do something nontrivial can also cause a massive efficiency hit to guarantee the correctness of garbage collection, so such overriding should only be done if really necessary to ensure that important resources held by the object are certain to be released.

3.9. Overriding the `equals` method

- By overriding the `equals` method in a subclass, you can implement a more specific notion of equality than the trivial **memory address equality,** the strictest possible equality relation.
- As a rule, your `equals` method should implement a mathematical **equivalence relation** so that it is **reflexive**, **symmetric** and **transitive**, splitting the space of possible objects of that type into disjoint **equivalence classes**.
- However, such **semantic properties** of methods cannot possibly be checked and enforced by the compiler, since the problem of checking whether an arbitrary method has any nontrivial semantic property is known (since the 1930's by the work of **Alan Turing**) to be **algorithmically undecidable** in the general case. (For some particular simple methods, their having the given semantic property can be reasoned mechanistically, but no algorithm can possibly do this in the general case and always produce the correct answer for every possible method.)
- There exist infinitely many possible equivalence relations within the same type, forming a mathematical **lattice**. On one end, we have the strictest possible concept of equality where every object is equal only to itself. On the other end, we have the loosest possible concept of equality where every object is equal to every other object. Between these there is an infinity of other equivalence relations, from which you have to choose the one suitable for your intentions.

- Overriding `equals` in some silly way that makes it not be a proper equivalence relation will cause other classes and methods (such as the `Collection` methods `add` and `contains`) that rely on these properties to behave in an unpredictable fashion. (If you try to build on quicksand, soon enough you will sink into it.)
- (Even if `equals` were written to implement a proper equivalence relation, methods that are not **congruent** with that particular sense of equivalence could act up strangely.)
- As the first pitfall, note that the parameter type of the `equals` method must be `Object`, since in Java, **any object can be equality-compared with any other object**. Comparing the proverbial apples and oranges is not any kind of compile time or runtime error, but a perfectly legal and meaningful operation whose answer in this situation just happens to be "no".
- (It is a surprisingly common misconception even among educated people that only questions that both (a) are non-trivial and (b) whose answer is "yes" are somehow finger-quotes "legitimate". This might stem from the difference between "ask" cultures versus "guess" cultures.)
- If you accidentally make the parameter type to be the same as the class in which you override this method, you are not overriding but overloading! The other classes will then be calling the original superclass version instead of your intended new version of the method. Again, use the `@Override` annotation to reveal such errors.
- Whenever you override `equals`, you should also override `hashCode` so that whenever `a.equals(b)`, also the two hash codes are equal so that `a.hashCode() == b.hashCode()`. Otherwise any collections that use hash codes to organize the stored elements (most importantly, the collection implementation `HashSet<E>`) can not work correctly.
- To ensure this property, the `hashCode` method may depend only on fields that affect object equality. Otherwise it will consider two objects distinct due to an internal difference that does not affect the `equals` method.
- Some unequal objects will also inevitably end up having the same hash codes, since there exists an infinity of potential objects that one could create, but only 2^{32} possible `int` valued hash codes to distribute among them. Any hash table data structure could therefore in theory, with truly astronomical levels of bad luck or if your worst enemy gets to create the objects so that they all have the same hash code, devolve into essentially a linked list.
- The hash code for an object should be computed by suitably scrambling and mixing the hash codes of **precisely** those fields whose values can affect the result of `equals`. If some field that does not affect the equality comparison is used in hashcode calculation, two objects that are equals will end up with a different hashcode, breaking the behaviour of all data structures and algorithms that make decisions based on these hash codes.

3.10. Dynamic type inference

- Since more specific types of equivalence make sense only within the same type, overriding `equals` properly requires us to be able to dynamically find out whether the given object is of some particular type.
- The operator `obj instanceof Bird` checks at runtime whether the object `obj` is a `Bird`, or any subtype of `Bird`. This operator also correctly gives a `false` result if `obj == null`.

- Alternatively, you can use **reflection** to ask an object its **exact type** with the method `getClass()`, natively defined in `Object` and inherited from there to all classes.
- To access the subclass members that do not exist in the superclass such as `Object`, you need a **downcast** to create a new subclass reference that points to the same object.
- All downcasts are **silently checked at runtime**, so it is not possible to use them to trick the type system to call a method that does not exist in the object.
- In Java, it is (almost) never necessary to use an **upcast**, since a variable of superclass type can already point to any object of any subclass type. However, an upcast exists in the language as it is needed to disambiguate between the calls to two overloaded methods when the offered argument would be compatible with both of their parameter types. (For example, the argument `null`.)
- (Speaking of `null`, this constant in the Java language is the only value of its own special type that is the opposite of `Object` in that this type is a **subtype** of all other types, therefore allowing any reference variable of any type to be assigned to have the value `null`. Since `null` is not an object, this does not lead to a contradiction or a paradox.)
- In practice, using `instanceof` (or the result-equivalent **reflection mechanism**) in practical code is usually **the absolutely wrong way to do things**, especially when it creates the infamous object oriented programming **antipattern** known as the **instanceof ladder**.
- Instead of manually dispatching the execution to different branches based on the inspected type of the argument object, you should always let Java's dynamic binding do this work for you!
- "*Anytime you find yourself writing code of the form "if the object is of type T_1, then do something, but if it's of type T_2, then do something else," slap yourself.*" -- Scott Meyers
- **Liskov Substitution Principle, bumper sticker equivalent corollary**: **If your polymorphic method cares about the exact subtypes of its argument objects, either your method or the entire design of your class hierarchy is wrong to begin with!** Go back to the drawing board, stat! Danger, danger, Will Robinson! Arooga! Arooga! Do nothing else until you find and fix the source of this most horrendous of all possible code smells in object oriented programming.
- This principle does not apply to **metaprogramming,** such as writing the `equals` method where it must be possible to talk about the type system and the exact type of some object. (Reflection creates conceptual paradoxes such as this one. For more of this sort of stuff, read the modern classic "*Gödel, Escher, Bach: An Eternal Golden Braid*".)

3.11. Decorator pattern

- Using a simple method overriding to express variability within some concept (for example, the difference between how various birds fly) is powerful, but would be completely absurd in some situations by forcing us to constantly repeat ourselves throughout the inheritance hierarchy.
- **Method overriding can only be used to express variation within methods that were originally included in the superclass.** However, it is impossible for any superclass architect to foresee all the future needs and ways that other people will be using and extending this class hierarchy, and the properties they would need the class to model and represent.
- For example, in the `Animal` example class hierarchy we want to allow the possibility for an animal to be **loud** so that a loud animal speaks in all uppercase. If we already have a `Cat` from

which we extend `LoudCat`, we would have to write essentially the same code for all other types of animals of which we want to also create loud versions.

- Even worse, allowing an animal to also possibly be mirrored would force us to have four separate classes for `Cat`, `LoudCat`, `MirroredCat` and `LoudMirroredCat`. With respect to the number of such properties that can vary independently, the number of subclasses would grow exponentially!
- A **decorator** (also called a **wrapper** in some other materials) is a subclass **from the same class hierarchy** whose objects make no sense existing on their own, but are placed "in front" of existing objects of that type. This **underlying object** is given to the decorator as a constructor argument, and stored there into a `private` field.
- After creation, the decorator object represents the underlying object, as far as the rest of the program is concerned, analogous to how an attorney represents a client in legal proceedings. **All access** to the underlying object should then always be done through the decorator, including method calls, passing the (composite) object as parameter to polymorphic methods, etc. (This is not a law of nature, but it is hard to see how you could ever go wrong by following this law.)
- The methods of the decorator generally call the corresponding method of the underlying object, but somehow modify the arguments down the line or the result up the line.
- Since the decorator type is itself a subtype of the original hierarchy, the decorated object can itself be further decorated to create arbitrarily complex computation pipelines, and express arbitrary variation within the same kind.
- The only real downside of using decorators is the **self problem**: since the object itself does not know that it has been decorated, during the execution of its methods that call some outside methods that require the object to pass itself as one of these arguments (such activity would be common in **callback methods**), this object does not know to pass the head of its decorator chain as this argument, but will pass itself. This can lead to confusing bugs when the same object is used both directly and through the decorator chain from different parts of the same program.
- A class that behaves otherwise like a decorator but comes from a different class hierarchy than the underlying objects that it decorates (usually because it has a different public interface than the original class hierarchy) is called a **facade** or an **adapter**.
- For example, `Scanner` is a handy and powerful adapter that offers a unified interface to any `String`, `File` or `Reader`, allowing us to read text from all these sources the exact same way without having to care whether the underlying object is a string, file or a character stream.

3.12. Constructors and inheritance

- Even though constructors are methods (albeit special in that you never call them yourself, but the JVM automatically calls them whenever it creates an object), they differ from other methods in that they are not inherited by subclasses.
- Even if the superclass has a dozen overloaded constructors, its subclasses have only the constructors that are explicitly written in them. You cannot therefore assume that subclass objects can be created the same ways and parameters that the superclass objects can be created.

- As before, if no constructors whatsoever are defined in a class, the compiler automatically synthesizes a do-nothing **default constructor** that takes no parameters. But as soon as you write even one explicit constructor, this is no longer done for you.
- Even though constructors are not inherited, they behave in a special way at object creation.
- When an object is created, **the default constructor of all of its superclasses** are **executed down the line** starting from `Object`. The constructor arguments, if given with `new`, are passed only to the appropriate constructor of the actual class itself, even if some superclass happened to have a constructor that could take those given constructor arguments. (Nothing guarantees that the superclass and subclass constructor, despite taking parameters of the same type, would use the given arguments in a semantically identical fashion.)
- Intuitively, every object starts out its lifetime as a raw `Object` and "grows" into the actual subtype during these stages of construction. The default constructor at each level initializes the fields that were defined at that level, so the constructors of subclasses only need to initialize the fields that are defined down there.
- (The subclass constructors can still re-initialize inherited `public` or `protected` fields, though, should that be needed. But of course we never use `public` or `protected` fields, but provide proper accessor and mutator methods for `private` data instead, yes?)
- **Field initializers** are executed before the constructor at each level. If you initialize a field with a method call (although it would be better if you didn't, just to avoid such order dependencies), be **very careful** not to call any method before the fields that it uses have been fully initialized. Otherwise these fields still have zero value when that method uses them.
- There is no way to detect during the construction whether the execution of the constructor chain has reached the actual subclass that was given to `new` and from which the object was created.

3.13. Overloading vs. overriding: technical interlude

- **Overloading**: the same class has several versions of some method, with the same name but different parameter types. (Access modifier and return type can vary over these methods.)
- **Overriding**: the subclass redefines a method that it inherits from a superclass, with the exact same name and parameter list. (Access cannot be more strict, return type can be more specific.)
- In a method call of the form `x.foo(y)`, **overloading is resolved at compile time** based on the compile-time type of the **variable** `y`. (Since objects do not exist at compile time, no decisions can be made during compilation based on object type!)
- In a method call of the form `x.foo(y)`, **overriding is resolved at run time** based on the type of the **object** that variable `x` refers to. (Since the compiler can't know the runtime type of the object `x`, it can't possibly bind the method call at compile time.)
- An important exception to the previous rule is that if the method `foo` is `private`, the call is always bound at compile time, seeing that it must be the version in the same class that the call itself is made, since `private` methods can be called in the same class only! This must be done this way, otherwise you could accidentally "override" a `private` method of a superclass, and the whole concept of `private` would have no meaning or purpose.

- Calls to `final` methods can also be bound at compile time, since we know that its subclasses will not provide a different version of that method. But this is only for efficiency, as the program would still work exactly the same way without such early binding.
- In calls of the form `super.foo()`, the call is also bound at compile time. This must be so, otherwise this call wouldn't work as intended whenever the method that contains this call was executed for an object created from some derived subclass further down the line.

Lecture 4: The Java Collection Framework

4.1. Interfaces as perfectly abstract superclasses

- In the theory of object-oriented programming, **multiple inheritance** means that a class can simultaneously extend **two or more immediate superclasses**.
- Any language that allows multiple inheritance must somehow resolve the can of worms that opens up **when the subclass inherits the same method or field from two of its immediate superclasses**, especially if these immediate superclasses themselves inherit this method or field from a common superclass of their own (the so-called **diamond problem**).
- For this reason, Java allows only a limited form of multiple inheritance where the class `extends` only one actual superclass, but `implements` as many **interfaces** as needed.
- An interface is an abstract class that has **no fields, and all its methods are abstract**. This restriction solves the above ambiguity of multiple inheritance.
- The distinction between classes and interfaces exists only in the Java language in a rather redundant fashion. In the actual Java type system, **there is no distinction whatsoever between classes and interfaces**, but **interfaces really are abstract classes and behave as such**. Period.
- An interface serves as a practical guarantee that **the class that implements it will have all the methods defined in that interface**. This is sufficient for writing polymorphic methods whose parameter type is that interface, since this allows the compiler to verify that the method calls made to that object inside the polymorphic method are correct with respect to the type system.
- Interfaces are the Java equivalent to the Pythonic spirit of **duck typing** in which the actual type of the argument object given to a polymorphic method is itself irrelevant as long as that object has the methods that the polymorphic method needs it to have.
- At runtime, the calls to the methods defined in the interface are dynamically bound to correct subclass versions just as before.
- Whenever a class implements some interface, all its subclasses will also automatically implement that same interface without you having to explicitly say so.
- Java 8 relaxed the definition of an `interface` slightly (well, quite a lot) in that methods no longer need to be fully abstract, but the interface can provide a **default implementation** for any method. If the class that implements an interface does not implement that method or inherit an implementation from its actual superclass, the default implementation is automatically used.
- Default implementations also make it possible to add new methods to an existing interface without breaking all the existing classes that implement the old version of the interface. This was rather necessary in creating the new advanced **computational streams** of Java 8.

- In a situation where a class inherits a default implementation from a superclass and an interface, the implementation inherited from the superclass is used. To resolve the ambiguity from inheriting a default method from two interfaces, the class must implement that method itself.

4.2. The Java Collection Framework

- The **Collection Framework** in the `java.util` package is a highly useful and educational example of class inheritance and polymorphism.
- Collection classes represent **dynamic aggregations** of elements, and provide three fundamental methods `add`, `contains` and `remove`, plus a couple of other methods built on top of these.
- From the root interface `Collection<E>` that defines the behaviour that all collections have in common, the collection hierarchy branches in two directions `List<E>` and `Set<E>`.
- `Set<E>` collections behave like a mathematical set in that duplicate elements are not allowed (repeated additions of an already added element are not errors, but simply do nothing) and that there is no internal structure shown to the outside, regardless of which data structure is actually used to store the elements to make the dynamic set operations more efficient.
- The most important `Set<E>` implementations are `TreeSet<E>` based on **balanced binary search trees**, guaranteeing O(log *n*) worst case running times for both dynamic set operations and **order-based operations**, and `HashSet<E>` based on **hash tables**, whose three dynamic set operations are O(1) on average, but no order-based operations are supported. (Theoretically these operations can degenerate to O(*n*) linear time slowdown, but the probability of this happening in practice is far too small to lose any sleep about.)
- `List<E>` allows duplicate elements, and internally keeps the elements in some linear structure that can be **indexed** with methods `get` and `set`, familiar from `ArrayList<E>` from CCPS 109.
- Adding and removing an element inside an `ArrayList<E>` is an O(*n*) operation, since the underlying array is not allowed to have gaps inside it. For algorithms that need to add and remove elements to arbitrary positions of the list, especially to the beginning which is the most inefficient case for `ArrayList<E>`, consider using `LinkedList<E>` instead.
- However, the random access methods `get` and `set` for `LinkedList<E>` are O(*n*) operations. Furthermore, `LinkedList<E>` needs more memory than `ArrayList<E>` to store the same elements, since it needs to store a separate **node object** for each stored element, instead of merely maintaining an array of references to those objects.
- If the element type `E` is an `enum`, `EnumSet<E>` can store the set efficiently as a **bit vector** with guaranteed O(1) dynamic set operations implemented using **bitwise arithmetic**.

4.3. Operations on collections

- The interface `Collection<E>` also offers various utility methods in accessing the elements in bulk. All concrete implementations also come with decent implementations of the `toString` and `equals` methods, unlike the primitive arrays of Java language.
- Like the utility class `Arrays` for primitive arrays, `Collections` offers a whole bunch of `static` utility methods for collections for common operations such as **sorting**, **shuffling** and finding the **minimum** and **maximum** element.

- For the `Collection<E>` implementations to work correctly when using your class as their element type `E`, it is essential that the `equals` method of your class implements a mathematically proper equivalence relation. If this semantic requirement is violated, the behaviour of the collection for such elements becomes undefined.
- For **order-based collections** such as `TreeSet<E>`, the element type `E` must implement the interface `Comparable<E>` that guarantees the method `int compareTo(E other)` that is used for element comparisons. The sign of the integer answer gives the result of the order comparison. The absolute magnitude of the returned integer is irrelevant, only the sign matters.
- If the element type `E` does not implement `Comparable<E>`, the set can be given a `Comparator<E>` strategy object to perform the required element order comparisons.
- For the same reason, elements added to collections should not silently change their internal state in any way that affects their `equals`, `compareTo` and `hashCode` methods. Immutable classes therefore once again offer great benefits for programming over mutable types.

4.4. Iterators

- To **iterate** over the elements of a collection, you can create an **iterator** to its beginning. An iterator acts like a smart finger that points to some element inside the collection, and can **advance** (or possibly also retreat, for the subtype `ListIterator<E>`) one step to the `next` element.
- Same way as in Python, functions can receive an iterator as their argument, and the iterator hides the actual type of the underlying entity that is producing the actual elements. This way, we don't need to write multiple functions to do the same thing with different types of sources of data.
- Somewhat strangely, the iterator method `next` returns the current element, and simultaneously makes the iterator silently step to the next element. In almost all other programming languages, these two operations are sensibly provided as separate methods.
- You can create any number of iterators pointing to the elements of the same collection. All those iterators can advance independently of each other, instead of being forced to advance in lockstep.
- Iterators allow us to write nice polymorphic methods that can operate on any `Collection` or `Iterable` without having to care about the exact subtype of that collection. This collection could even be **virtual** or **infinite** so that the iterator itself dynamically produces the elements of this virtual collection one at the time, the method using this iterator being none the wiser.
- Each iterator knows how the data is organized in the underlying collection, and hides all these pesky details from its users. Your polymorphic methods that receive iterators as arguments should never have to care what the underlying collection is that produces the elements.
- Iterating through an implementation of `List<E>` in the Java standard library is guaranteed to produce the elements in the order of their indices. The order in which the elements of `Set<E>` are produced depends on the internal implementation of the set data structure and its iterators. The subtype `SortedSet<E>` adds the guarantee that iteration goes through the elements in sorted order, and `NavigableSet<E>` further refines this with many useful order-based operations.
- Iterators have an optional method to `remove` the element that was previously returned by the method `next`. Calling this method will succeed only if the underlying collection supports the element removal. This method can be called only once following a call to the method `next`, otherwise an `IllegalStateException` is thrown.

- Even though the handy **for-each loop** acts as **syntactic sugar** for all iterators instead of only for arrays, you must perform this iteration explicitly if you want to use the `remove` method.
- **Modifying a collection in any other way while it is being iterated is not guaranteed to work**, and depending on the actual subtype of the underlying collection, can either work correctly, cause strange behaviour, or cause a `ConcurrentModificationException` being thrown.

4.5. Strategy pattern

- Inheritance and method overriding are completely static and set in stone at compilation. The decorator pattern is fully dynamic, as objects can be decorated in different combinations on the fly during the program execution.
- As handy and flexible the decorator pattern is, it would also be equally absurd in some situations. For example, suppose that you have some collection of elements whose elements you want to sort or analyze according to various different criteria, not just the one that is baked into the data type of those elements.
- Since the objects are stored privately inside the collection that is given to you, you would not be able to decorate them for modified comparison purposes. Instead, the designers of the `Collections.sort` method were prepared for this often occurring need by allowing you to give it a `Comparator<E>` strategy object that performs the individual comparisons between objects of type `E`.
- Similar `Comparator<E>` strategy objects can also be given to `TreeSet<E>` to perform the element order comparisons needed to organize the underlying binary search tree.
- An object that is given to another object or method to “fill in a gap” that has been left open inside it is called a **strategy object**. The same object or method can then be made to dynamically act in different ways simply by giving it a different strategy object, following the flexible principle of **dependency injection**.
- The component classes of the **Swing GUI library** next week will also heavily use various strategy objects to allow configuring the details of their behaviour, for example `Stroke`, `Color`, `LayoutManager`, `Font`, `Border`, ...
- In the decorator pattern, the underlying object and the decorator object both come from the same class hierarchy. In the strategy pattern, the strategy object and the object that this strategy object is given to come from different class hierarchies.
- The strategy object, same way as a decorator, can easily be programmed to do some internal bookkeeping or **logging**, which allows us to “crack open the hood” to take a peek inside the object that this strategy object is given to.

4.6. Java Collection Framework as a teachable moment

- Even though all `List` implementations support the indexing methods `get` and `set`, these are not generally guaranteed to operate in O(1) time. The implementations that guarantee this, for example `ArrayList`, implement the **marker interface** `RandomAccess` allows polymorphic methods that receive lists as arguments to change their behaviour depending on whether the argument supports random access operations.

- A marker interface has no methods, but is essentially one immutable `boolean` value embedded directly into the type system, available both at compile time (as opposed to storing this value to a `static final` variable inside the class) and runtime using type inspection with `instanceof`.
- (Marker interfaces are now semi-obsolete, as **annotations** should be used in their place. Marker interfaces have the advantage of serving as a parameter type of a polymorphic method, though.)
- This is, of course, a horrendous violation of the principle that a polymorphic method should never have to worry about the exact subtypes of its argument objects. Yet any polymorphic method that operates on `List` arguments has to be prepared for this possibility, unless its logic can be implemented with sequential iteration without `get` and `set`.
- The only way to find out whether the given iterator supports element removal is to call its method `remove` and see what happens. (The framework designers did not deem necessary to use a marker interface analogous to `RandomAccess` to denote this important piece of knowledge.)
- Even then, nothing can tell you whether given iterator `remove` works in guaranteed O(1) time (as it does in `LinkedList<E>`) or O(*n*) (as it does in `ArrayList<E>`), which surely makes it hilarious to write polymorphic methods that use `remove` on an iterator received as argument.
- There now exist branches of `Queue<E>` and `Deque<E>` whose very existence as part of this hierarchy is absurd to begin with, so we shall ignore them in this course except as a warning example of how not to use class inheritance. Saying that some `Queue` is a `Collection` is exactly as silly as saying that some `Car` is an `Engine`!
- Queue implementations merrily violate several postconditions that were explicitly promised by methods in the `Collection` interface, such as the rather important guarantee that calling `add` can never fail unless adding an element to a set that already contains it, or if the entire JVM runs out of heap memory. Now any polymorphic method that quaintly relies on this promise will crash or do something unpredictable when its argument is a queue backed by fixed size storage.
- (Instead of `add`, use the method `offer` to try to add an element to the queue.)
- Queues, stacks and deques are super handy and useful in many computer science algorithms, though, don't get me wrong! **They just simply aren't collections**, and in any properly designed class library for data structures, would form a completely separate class hierarchy of their own.

4.7. Association maps

- Each collection stores individual keys, whereas a **map** (or a "dictionary" or a "hash") stores **associations** from **keys** to **values**. The `Map<K,V>` interface therefore uses two generic type parameters, `K` for the type of its keys and `V` for the type of its values.
- Thank goodness, `Map<K,V>` is a completely separate class hierarchy from `Collection<E>`. (Somebody had more sense back then than the creators of that blasted `Queue<E>` branch.)
- The dynamic map operations `get`, `put` and `remove` are analogous to set operations `contains`, `add` and `remove`, but operate on key-value pairs instead of individual keys. (The methods are intentionally named differently from those of `Set<E>` so that there is no confusion whether we are currently operating on a set or a map.)
- The underlying data structure is a set of `Entry<K,V>` pairs where the set operations are performed on the key. Finding the key and its associated value is efficient, but finding a value and its associated key is generally a linear time operation. (Analogy: using a dead tree phone book, it

is easy to use **binary search** to find the given person by name to look up their phone number, but the **reverse lookup** of finding the person who has the given phone number can be done no better than linear search.)

- The method `keySet` returns a `Set<E>` **live view** to the underlying map. These views share the underlying data structure of the map, so any modification done to one will also affect the other.
- In a map, **each key can be associated with at most one value**. Trying to associate another value to a given key using `put` removes the previous association, returning the old value as the result of `put`. To simulate a `Multimap<K,V>` where each key can be associated with multiple values, implement it either as a `Map<K,List<V>>` or a `Map<K,Set<V>>`).
- In Java 8, additional `default` methods `replace`, `getOrDefault` and `putIfAbsent` are often useful, and more efficient than first checking if the map contains the given key and then searching for that same key again to access its associated value, since these methods can now internally perform only one data structure search instead of two to find the key.
- (Side exercise: Analyze the following optimization to speed up this common case of searching the same key twice: have the `Map` implementation cache the location of the result of its most recent search, and before searching for a key, check if that key is equal to the cached result to make such consecutive searches of the same key to work in O(1) time. Yay or nay?)
- Normally when counting how many times the given elements occur in the data that you loop through, you use an array of integer counters. But if you don't know beforehand what the element values are going to be (for example, you want to count how many times each word occurs in some big text such as *War and Peace*), use a **counter map** of type `Map<K, Integer>` instead. Then for each element, use `getOrDefault` to find out how many times that element has been encountered so far, and `put` that value plus one back to the map.

Lecture 5: Swing

5.1. AWT and Swing

- **GUI programming** was historically the first "killer app" of object-oriented programming in the eighties with the **Smalltalk programming language**, making many things trivially easy that would be difficult to achieve with traditional imperative programming.
- Many techniques and ideas of computer science are at least a decade or two older than even most working programmers would assume them to be, and many such ideas had to wait patiently for the hardware processing power to catch up with their potential.
- In early versions of Java, the **Abstract Window Toolkit** class library provided **hooks** for the native GUI components of the underlying window system. Therefore GUI in both Java **applets** and **applications** was different and inconsistent across different environments, and limited to only to capabilities that all GUI engines have in common.
- To remedy this, **Swing** was built on top of AWT, reusing and extending its non-component classes (e.g. `Color`, `Font`, `Image`) and replacing its component types (e.g. `Button`, `TextField`) by its own component types (e.g. `JButton`, `JTextField`), rendered by Swing on three basic AWT components for a consistent **look and feel** on all platforms.

- Instead of using native components, Swing basically just **draws a picture of each component.** But for the human user, a picture of a component is just as good as a real component, as long as it reacts to all user actions the same way as a real component would! "If it walks like a duck..."
- Fundamentally, inside the computer there is only the processor that moves bytes around and performs simple computations on them. Everything on the screen is just pixels, and the higher-level division of these pixels into semantic-level entities is done by the human who is looking at the monitor. The mindless machine itself is not aware of any such distinctions.
- The more experienced the computer user, the more they think about computers using high-level semantic distinctions. For such a user, the drop-down menus on top of the OS X screen are essentially different from the drop-down menus rendered by the web browser inside the web page that is being displayed.
- In a curious zen-like spirit, a complete beginner has a more realistic view of the computer as a whole so that they don't see any distinction between these two types of drop-down menus!
- Swing is an entire **framework**, not merely a library. To paraphrase the comedy stylings of Yakov Smirnoff, "*When using a library, you call the methods. When using a framework, the methods call you! What a language!*"
- Names of all Swing component classes start with a capital `J`, such as `JTextField` and `JButton`, to make it clear whether we are talking about a Swing or AWT component.
- A hefty number of common GUI component classes are included in `javax.swing`, and you could extend any one of them to further extend and customize its behaviour. In this course, my examples always extend `JPanel`, a handy little component that doesn't really do anything on its own, but acts as a blank canvas on which we can impose our own behaviour.
- All Swing components extend the abstract superclass `JComponent`, a massive class filled with all sorts of bells and whistles. However, Swing is well designed so that everything usually works fine "out of the box". For the purposes of this course, Swing is also an excellent example of various object oriented programming techniques and design patterns.
- The **only setting that you must provide** for every Swing component is its **preferred size**, as Swing cannot possibly read your mind to divine whether you are creating some little button or an entire screen-sized splash.
- Swing components can contain other components inside them. A `LayoutManager` **strategy object** positions and resizes the contained components for you. You can also set this manager to be `null`, and manually **free position** every component yourself with pixel precision. (Or in real life, use a special graphical GUI maker tool for this purpose.)
- Only the Swing **top-level containers** `JFrame`, `JApplet` and `JDialog` (and therefore all their subclasses) have the ability to exist on the user's screen on their own. All other Swing components must reside inside some top level container to show up to the user. (These objects still exist and work in the memory on their own the same way as all other objects, as they are not in any way magical.)

5.2. Graphics rendering

- At any time, Swing can arbitrarily decide that some component needs to be redrawn. The components have no control over this, and **must at all times be ready to draw themselves** on command the way that they look like at that particular moment.
- In general, Swing is always the boss in charge and can do whatever it wants. This is the only way that components written by different people in different times can work smoothly together. (If two components try to boss around Swing simultaneously, what would happen?)
- The method `paintComponent` is called by Swing, passing a `Graphics` object as a sheet of paper to draw on. Once this method returns, Swing posts the `Graphics` object on the user screen. By definition, that is **what that component looks like at that moment**.
- Since the original `Graphics` class is very primitive so that it essentially has the capabilities reminiscent of some 1980's home computer Basic, these days a more powerful `Graphics2D` subclass object is guaranteed to be always passed to this method as its argument. This class provides a rendering engine with the capabilities that we expect our graphics rendering to have in this millennium.
- You need a **downcast** to call the new methods of `Graphics2D` that don't exist in `Graphics`. Calling the methods that are already in `Graphics` requires no downcast, since dynamic binding is still in full effect to bind these calls to the `Graphics2D` overridden versions at runtime.
- The coordinate system of the component has its origin (0, 0) at the top left corner, and the **y-coordinate grows downwards**, following the usual computer graphics convention where the familiar coordinate system of mathematics textbook has been mirrored with respect to *x*-axis.
- Even though integer coordinates correspond to the pixels, rendering is performed using higher precision `float` resolution, which can affect colours and **antialiasing**, and can also make a difference if the coordinate system is **transformed**.
- Doing all position calculations using floating point decimal precision rather than integers is especially important in **calculating animation paths**, even if the object point coordinates in each animation frame then get truncated to the integer coordinate resolution in rendering. The more fine-grained the motion, the larger the difference in the end result. Forcing all intermediate values to be integers introduces **quantization artifacts** analogous to aliasing effects.
- **Geometric shapes** are represented as objects from subclasses of `java.geom.Shape`. This is an entire powerful class hierarchy of its own, working behind the scenes. The shape objects reside on the infinite two-dimensional plane, and don't even know anything about Swing and its components that render their visual representations. Swing components then give our human eyes a fixed-sized peephole showing some finite part of this infinite two-dimensional space.
- To control the drawing, `Stroke` and `Paint` strategy objects can be given to `Graphics2D`.
- To create more advanced combo shapes, use the `Area` decorator that allows shapes to be modified with combinations of **affine transformations (translate, rotate, scale)** and **constructive geometry operations union, intersection and difference**. Arbitrarily complicated geometric shapes can be created this way.

5.3. Nested classes

- From 1.1, Java has allowed classes to be **nested** inside each other. If the class `B` is nested inside class `A` without using the modifier `static` (thus making `B` an **inner class**), **the objects of `B` cannot exist on their own**, but only in the **context** of some object of the **outer class** `A`.
- Therefore, a good way to determine whether `B` should be an inner class of `A` rather than a top level class is to simply ask “Would it make any sense for an object of `B` to exist on its own, or is it innately tied to the framework of class `A` from which it cannot logically be separated from?”
- The practical consequence of class nesting is that an object of `B` can only be created inside the instance methods of `A`. The object `this` will be the context object of the `B` objects thus created.
- In the heap memory, each object of type `B` contains **an extra unnamed reference** to this context object. Through this implicit reference, the methods of `B` can access all members of `A` (even those that are `private`) directly as if they were its own, no special syntax is needed.
- Nesting a class even further inside a method also allows the **local class** methods to access all effectively `final` local variables of the surrounding method. (Why only `final` ones? Because the local class object can still escape from inside the method and **continue to exist after the local variables of that method have vanished from the stack**. Therefore the JVM cheats a little by storing **defensive copies of the local variables** inside the local class object. If these variables are `final`, it doesn’t matter whether you use the original variables or the defensive copies.)
- Making a nested class `static` (or defining it local inside some `static` method) allows those objects to exist on their own without the context object. Of course, the nested class methods then cannot access the outer class instance methods, since no context instance exists.
- If the nested class `B` contains a field with the same name as another field in the outer class `A`, the special syntax of `A.this` inside the methods of `B` refers to the context object. In these methods, just `this` is always the nested class object itself.
- Theoretically, the level of nesting could go arbitrarily deep (illustrating how the **zero-one-infinity** principle applies to not just programming, but programming language design), but in practice never more than two. (Similar infinite ladders of which we only ever use the first few steps exist elsewhere: for example, the number of dimensions of multidimensional arrays.)
- Non-static nested classes are not allowed to have `static` fields, unless they are `final` named constants that are initialized at compile time. See the Stack Overflow page "Why does Java prohibit static fields in inner classes?" for more discussion.
- Java compiler will turn every class into a separate bytecode file. Since the dollar sign character cannot be a legal character inside a Java identifier, the filename uses dollar signs as separators for nested names, such as `Foo$Bar.class` for the nested class `Bar` defined inside outer class `Foo`.
- It is possible for an inner class to be **anonymous**, albeit with a rather strange syntax. The compiler will name these anonymous inner classes with integers, such as `Foo$1.class`, `Foo$2.class` etc.
- The **lambda operator** (see later) is often used to turn anonymous inner class definitions into one-liners by making the compiler fill in all that rigmarole of class and method names.

5.4. Object factories

- If the constructors of some class are declared `private`, no objects of that class can be created anywhere from outside that class.
- Instead of defining a `public` constructor, the class defines one or more **static factory methods** to create objects of that type. For example, see the Swing strategy class `Border` and the corresponding factory class `BorderFactory` for these methods.
- A factory method has several advantages over a constructor. For example, the factory method can sometimes quietly return an object of some subclass type, or an arbitrarily complex decorated version of the object, the caller being none the wiser about this but happily using the object as if it were of the original type.
- **If the objects are immutable**, the factory can keep track of the objects that it has already created, and if an identical object already exists, return a reference to that existing object instead of creating another, redundant identical object.
- For mutable types, the class can also allow a mechanism for the outside world to give back objects that are no longer needed, and store them internally in an **object pool**. The next time the factory is requested to create a new object, it can recycle an old object from the pool and simply re-initialize its fields to the new values.
- This mechanism can be quite valuable in a class that implements some data structure such as a **binary tree** or a **hash table** that constantly have to create and release little **node** objects that make up these structures by pointing to each other along with the **payload data** that they carry.
- Especially in applications that constantly create and release small objects, this can produce huge time savings and even prevent the application from running out of heap memory when the new little object creation and release is so frequent that the JVM garbage collector cannot keep up.
- The factory itself can also be an entire class hierarchy, with the superclass being an **abstract factory**. The canonical example is some game where the game engine receives a factory that creates new enemies in the game. Different levels of the game can then be defined using different subclasses of the enemy factory.
- An abstract factory is a strategy object that is consulted whenever a new object is needed. The algorithm that creates these objects therefore does not have to care about the specifics of how these objects come to life and what are the specific rules that control the creation.

5.5. Event listeners (Observer pattern)

- When you want to react to some event happening, **polling** can have its time and place, but it is usually rather inefficient to keep repeatedly asking whether something has taken place.
- **Swing event listeners** are a good example of **Observer pattern**, in which any component can register any number of listeners that get notified when an event takes place.
- Each event listener **implements the corresponding interface**, thus guaranteeing that the event listener object contains the appropriate methods to react to the events.
- The event listener objects wait silently without wasting any processor cycles, until they are actually activated by the call of the method that corresponds to the event that took place.

- The relationship between the components and the listeners is **many-to-many**, so that the same listener can listen to multiple components, and the same component can be listened to by multiple event listeners that will all react to that event.
- The best practice to implement the event listeners is to write them as `private` nested classes inside the component class that they are hardwired to listen to. Since the listener is written to listen to one particular component, it should not be exposed as `public`.
- For this reason, **the common shortcut of making the Swing component itself to be an event listener is just plain wrong and false advertising**. This component cannot possibly have any meaningful ability to react to those events coming from arbitrary Swing components.
- If the event listener does something that changes the way the component is supposed to look, call that component's method `repaint` to request Swing to redraw it at the next opportunity.
- Of course, there is no law saying that the event has to originate from a Swing component, since anybody can call the event handling methods of any object. For example, `Timer` objects generate action events at every tick of the metronome.

5.6. Turtle graphics

- The concepts of a **cartesian coordinate system and trigonometry** would be a bit too high level math for curious children of all ages from five to one hundred and five eager to learn graphics programming.
- **Turtle graphics** allow drawing operations to be described in a more intuitive way that a small child can understand and even act out by himself, since these operations are done with respect to **relative positioning** instead of the grid of absolute Cartesian coordinates.
- Conceptually, a **turtle** is an object that moves around on the two-dimensional plane, internally remembering its **current heading** (as an angle) and **current position** (as *x*- and *y*-coordinates). The users of the turtle usually don't care about this absolute position.
- In the tail of the turtle there is a **pen** that can be either **up** or **down**. When the pen is down, the turtle leaves behind a drawn trail as it moves.
- The turtle can be given two basic instructions: (1) **move** a given distance to its current heading, or (2) **turn** in place by a given angle. All higher level graphics such as drawing a square, regular polygon or a star can be done by suitably combining and repeating these two operations.
- A more advanced turtle can also remember its previous positions by **pushing** them in an internal **stack**, and be commanded to return to its previous position **popped** from the stack. This allows us to easily draw various **treelike** structures and **fractals** where multiple branches split into different directions from the same point.
- Classic turtle graphics programming in the style of the 1970's **Logo programming language** never talks about the absolute position, but after initialization, all movement is done using only the relative position operations.
- Traditionally, turtle graphics have been used to teach **imperative programming such as conditions, loops, procedural abstraction and recursion** in a very intuitive manner where the programming errors become visible as the drawing is different from what was expected. (In retrospect, that sort of instruction would have been suitable for the course CCPS 109.)

5.7. Turtles for the new millennium

- Here in CCPS 209, we shall use turtles to practice inheritance, polymorphism, dynamic binding, decorators and other central object-oriented programming concepts. In the instructor's example class hierarchy, the interface `Turtle` is a **facade** to `Graphics2D` that **translates** turtle instructions to the previously seen rendering primitives of `Graphics2D`.
- To make life easier for people who want to extend `Turtle`, the subclass `AbstractTurtle` provides a reasonable concrete implementation for all methods except `move`. (The subclass `BasicTurtle` then extends that method to draw a solid line exactly as it is told.)
- Whenever some interface has lots of methods, it is a good idea to provide such a utility class to make life easier for those who want to create their custom implementation of that interface, and provide some kind of reasonable implementation (it doesn't need to be maximally fine-tuned or optimal) for as many methods as possible.
- For example, the utility class `AbstractCollection` for the `Collection` interface implements methods such as `addAll` or `forEach` that can be implemented to use `add` and `iterator`, even if these methods are still `abstract` and the subclasses must then implement them.
- (In Java 8, you can also define `default` implementations inside that interface itself. However, this approach would not work in this example, since these method implementations also need data fields, which cannot be defined in an interface. This would work better for `MouseListener` and such, where the `default` implementation of each method just needs to exist and do nothing.)
- Concrete subclasses of `Turtle` represent turtles with different abilities and behaviour, such as a `HandDrawnTurtle` that leaves behind a line rendered in simulated "hand drawn" style. All these turtles are commanded with the same methods of the superclass `Turtle`. See the `TurtleDemo` example class for a demonstration of a spiral drawn this way, the same sequence of `move` and `turn` commands passed to six different turtles.
- Again, any polymorphic method that uses a turtle to do some drawing **does not, and should not, have to care which exact subtype of `Turtle` it is given as argument**.
- In several labs of this course, you will be extending and decorating the existing turtle classes in various ways to create your own customized types of turtles.
- Breaking down a move recursively into a series of smaller moves and turns makes it easy to render various fractals such as **Koch curve** or **Hilbert curve**. Such **fractalizing** is best done with decorators so that it becomes independent of other details of rendering. See the example classes `ZagTurtle` or `AntennaTurtle` that can decorate any existing turtle.
- When you write your own **fractalizer**, make sure that the **turns in each level of recursion add up to zero**, and the **subdivided moves add up to the distance that the turtle was supposed to perform**. Neither rule is a law of nature or man, but makes the result more aesthetic and well-behaved.
- **Lindenmayer systems** (also known as **L-systems**) are a classic algorithmic technique to generate a long and convoluted series of `move` and `turn` instructions for turtles to produce beautiful fractal shapes, especially those that simulate plants or **space-filling one-dimensional curves**. This theory is way past this course, but those interested can look at the example classes `LSystem`

(which is itself a facade for `Turtle`, which in turn is a facade for `Graphics2D`, which in turn is a facade to… who knows, something internal to the Java virtual machine!) and `LSystemDemo`.

Lecture 6: Exceptions

6.1. Pre- and postconditions

- Every method has a set of **pre- and postconditions** that describe **the intended semantics of the method by describing what it is supposed to achieve and in what circumstances**. The method implementation itself is encapsulated into a **black box** from the point of view of the caller who can only see the end results once the method has terminated.
- The pre- and postconditions are not expressed in the Java language since they are **semantic properties** that reside one level of abstraction above the language. Instead, they are written in natural language in the comments and (**JavaDoc**) documentation of the method.
- Same as any programming language, the Java language itself already describes perfectly **what will happen during execution**, whereas the pre- and postconditions describe the intention of **why those things happen**.
- **Always comment on the why, not on the what**. If your code is so complicated that you need comments to explain to other people (and yourself) what it does, that is a surefire sign that you should go back to the drawing board and completely redesign that code without trying to be super terse and clever.
- (In programming, calling something "clever" is not always a compliment.)
- **Postconditions describe what the method promises to achieve**, that is, what is guaranteed to have happened when the method execution is finished.
- Every method should have **at least one unit-testable non-trivial postcondition**, otherwise that method has no reason to exist. For a particularly notorious example of a method without any testable postconditions, check out `System.gc()`. What could possibly constitute a failure to pass the test after that method has terminated?
- (That whole silly method is pretty much the programming analogue of those "door close" buttons in elevators that are not actually connected to anything real.)
- Some methods can't possibly guarantee that they will fulfill their postconditions under all possible circumstances. The behaviour of the method may depend on properties of the outside world beyond the reach of the method. For example, a method that reads data from a file cannot possibly do anything meaningful if the file doesn't exist, or is not readable, or if the data inside it isn't properly formatted, or if the file contains semantically nonsensical information.
- **Preconditions describe what the method needs from the outside world to be able to work**.
- Unlike method postconditions, the set of preconditions may well be empty; **in fact, this would be a good thing**, since such a method can never fail to fulfill its purpose, unless the method implementation contains a bug.
- **It is not the method's fault that the outside world failed to fulfill these preconditions**. That is not a bug in the method, as the method has no divine powers to fix things to be right again.

- In principle, anything could happen if the caller fails to verify that the preconditions hold before the call. Typically, a method **throws an exception** to indicate that preconditions do not hold.
- The opposite principles of **design by contract** and **defensive programming** are both valid approaches in checking preconditions. The former minimizes code, the latter maximizes safety.
- Even when the superclass method is `abstract` and has no implementation, **it can still define pre- and postconditions** that all concrete subclass implementations of that method should adhere to. Because pre- and postconditions are semantic properties of the code, no compiler can algorithmically verify in general that the method body will make its postconditions true for all of the legal argument values of that method, so the responsibility for this is solely on the shoulders of the programmer.
- Be careful when designing the postconditions of the method, since unlike humans, the logic does not care about what you intend to happen. For example, the postcondition "The elements of the parameter array are in sorted order after the method has terminated" would be trivially satisfied by a method implementation that fills its parameter array with all zeros.
- Sometimes people really, really wish that the method had some additional postconditions. If the current implementation of the method coincidentally happens to satisfy those wished-for postconditions, the grim error of **relying on undocumented behaviour** will become painfully evident once the method implementation changes in the future so that the method no longer guarantees those same assumed postconditions, making all bets be off once again.

6.2. Liskov Substitution Principle

- As previously noted, **inheritance should model an is-a relationship between the two concepts** from the problem domain that those classes represent. However, since this is a **semantic property**, no compiler can meaningfully check or enforce this requirement.
- It is always possible to write nonsense in any language, no matter how high or low level. The only alternative to this would be to have a syntax and type system so massively restrictive that it warrants the term **"programming in a straitjacket"**, first historically applied to **Pascal** programming.
- Many classic languages such as Smalltalk or Pascal (or for that matter, these days also Java) had many good ideas that were revolutionary for the thinking and limitations of their era, but all these good parts have since been appropriated into more modern languages.
- The famous **Liskov Substitution Principle** (from hereafter, **LSP**) formulated by Barbara Liskov helps us to determine whether some class `Foo` should be made a superclass of another class `Bar`. Why can we say `Car extends Vehicle`, but should not say `Car extends Engine` ?
- LSP is not a law of either nature or man, and can be freely violated at will. However, the LSP is a SOLID **engineering principle** that makes your polymorphic methods work right.
- **If you want to write class hierarchies that allow writing meaningful polymorphic methods properly and safely, you simply must follow LSP**.
- Where LSP is not followed, chaos and disharmony will always eventually follow. Where LSP is followed, order and harmony will naturally follow.
- By following the LSP, you soon realize that you almost magically solve many problems before those problems even had a chance to emerge to bug you. All classes that you create just seem to

click together seamlessly, and the general nature of these connections allow you to easily experiment with new combinations of ideas that you never would have even thought of, had you been programming in some less organized imperative language.

- Oversimplifying only a little bit, LSP can be expressed as the requirement that **whenever you override a subclass method**, its **postconditions can't be made weaker** (but they can be equal or stronger), and its **preconditions can't be made stronger** (but they can be equal or weaker). This seems pretty simple and obvious, but like all great ideas, it was not simple and obvious until somebody had stated it explicitly.
- Intuitively, the subclass version of the method must work in any situation where the superclass method promises to work, and in any of those situations, achieve at least as much as the superclass method promises to achieve.
- If this principle is followed for all methods, **the subclass object can be substituted in place of a superclass object**, and the rest of the system still works just as well as before, or even better.
- Especially **the subclass object can be passed to any polymorphic method that expects a superclass object as its parameter**, and the polymorphic method works happily with that given subclass object.
- **LSP is both necessary and sufficient for polymorphism to work! Without it, there can be no meaningful polymorphism that would gain anything real!**
- Even more importantly, if the substitution principle can't be followed for even one method of `Foo`, that tells you that no matter how much your intuition might insist otherwise, you must resign to the stark reality that the class `Foo` cannot be a superclass of `Bar` to begin with. You should not impose the inheritance relationship on classes just to be able to inherit some particular method.
- Even though both `Cartoonist` and `Gunman` have a method `draw`, trying to combine these two under a new common superclass `Drawable` results in this superclass having a method `draw` with no meaningful postconditions that both subclass implementations of `draw` could fulfill. Hence the method has no reason to exist, and thus the interface `Drawable` also has no reason to exist.
- The most famous example of this phenomenon is the question of how the classes `Circle` and `Ellipse` should be related to each other in inheritance. Even though every circle is an ellipse, it turns out that neither class can be made to be a subclass of the other, assuming the existence of **mutator** methods that can turn a circle into a non-circle.
- This classic example shows that **the subclass relationship is a different thing from subsethood**, even though these often coincide and are occasionally mistakenly assumed to be the same thing.
- Once again, **immutable objects make everything easier to reason about**. If no mutator methods exist in the class, `Circle` can be a subclass of `Ellipse` the exact same way that `Car` is a subclass of `Vehicle`... which also requires the tacit assumption that there do not exist any mutator methods to attach wings and a jet engine to some car to turn it into an `Airplane`!

6.3. The exception machinery

- What should the poor method do when it is asked to do something for which it is flat out impossible for that method to achieve its postconditions?
- Older languages would have this method return a special **error code** value (typically negative one in the spirit of C, assuming that an actual result of that method cannot possibly be negative) or

change the value of a global **flag** to indicate an error, from which the caller is expected to realize that the returned result is actually meaningless nonsense.

- However, from a software engineering perspective, in many ways this is a defective approach. The caller would always have to check this flag for errors after every call. Whenever some error occurs, there is no easy way to propagate the error handling up the call chain to the level where the cause of error actually took place.
- This is especially true in library classes and methods that are intended to be used in many different applications, so that this library cannot possibly know anything about its caller and what exactly there ultimately caused this error to manifest, nor the steps needed to fix the problem.
- In Java and other modern languages, a failing method instead **throws an exception** to indicate a failure. **No result whatsoever is returned**, since **no meaningful result can logically exist**.
- An exception is **not any kind of special return value**, but **throwing an exception** is a separate mechanism from **returning a result**. Hence the intentional use of two different verbs.
- To terminate the method execution abnormally by throwing an `Exception` object, use the keyword `throw` instead of `return`.
- Different types of exceptions can be thrown to indicate different problems that can occur. This way, the caller knows how to handle these different problems in different ways. For example, a method that reads data from a file can fail because the file does not exist, or that it exists but contains syntactic nonsense, or that it exists but contains semantic nonsense. All these situations would probably be handled in very different ways by the caller.
- The method documentation should fully describe the possible exceptions that could be thrown by that method, and the preconditions whose violations that make them occur.
- Exception objects can also contain a human readable **message string** that explains what went wrong, to help the programmer to further identify and analyze the problem to debug it.

6.4 Catch or release

- Whenever a method throws an exception, the caller has a choice of two possible ways to proceed; they can either **catch and handle** the exception, or **let that exception fly** to the previous level of the call chain, as if this calling method had itself thrown that exception.
- Uncaught exceptions terminate the thread that threw them. JVM will then spit out the **stack trace** of that exception to the **standard error stream** of that JVM process. (In BlueJ, the standard error stream will show up in red text in the bottom of the console window.)
- The caller could also catch only some types of exceptions and let others fly on. In general, the caller should only catch those exceptions that it can meaningfully handle at that level.
- Be especially wary of the common **exception antipattern** of catching all exceptions with an empty handler, this way quietly sweeping your problems under the proverbial rug.
- To catch an exception thrown by a method, surround the block of code that contains the call in a `try` block. If no exceptions are thrown, the `try` block is skipped as if it weren't even there.
- The `try` block can be followed by one or more `catch` blocks for different types of exceptions, their bodies being executed if an exception of the particular type gets thrown.

- With or without catch blocks, the try block can also be followed by a `finally` block that is guaranteed to be executed before the execution leaves the preceding block, regardless of which way the execution exited the block and whether any exceptions were thrown.
- A common programming pattern is to acquire and use exclusive resources in the `try` block, and then release them in the `finally` block, to guarantee that that resource will be released.
- As a general rule, **you should not allow any exceptions to fly out of a `finally` block**. Since **a method can throw only one exception at the time** (just like a method can return only one result at the time), this second exception would fly out and the original (and obviously more important) exception would vanish into the night.

6.5. Checked and unchecked exceptions

- Unlike most other languages that have an exception mechanism to indicate logical errors, Java sharply divides its exception classes into two separate kinds, **checked** and **unchecked** exceptions.
- Checked exceptions **must either be caught**, or if the method lets them keep flying to the previous caller, **declared with the `throws` clause in the method signature**, so that compiler can enforce that the caller will also either catch or declare that exception.
- **Checked exceptions are used to announce that the caller did not fulfill the preconditions of the method.** Therefore the caller is held fully responsible for handling that exception, and the compiler will enforce this with the same strictness that it uses for the rest of the type checking, such as the correctness of the method return type or the parameter types.
- When overriding a method inherited from the superclass, **the subclass version may not declare any new checked exceptions** that were not already explicitly declared by the superclass version. (It can declare fewer of them, though, if you are certain that some particular exception will never be thrown by this particular subclass version of that method.)
- **A method that has no preconditions can't possibly throw any checked exceptions.** It simply has no excuse to fail, no matter what the caller and the rest of the outside world might have done before the method assumed control of execution.
- **Unchecked exceptions** of the Java library such as `ArrayIndexOutOfBoundsException` or `NullPointerException` indicate **a bug in the method itself**. These can also be caught and handled, but this is voluntary and not required by the language and the compiler.
- It would make no sense to require all user code to be able to catch and handle all possible bugs that can exist in all methods that they call. Usually the caller could not do anything anyway, even in principle. All computation implicitly depends on the assumption that the previous computation steps have achieved their promised postconditions.

6.6. The `Throwable` class hierarchy

- The root superclass of all exception classes is the superclass `Throwable`. Only objects created from the subclasses of this class can be thrown as exceptions. `Throwable` has two subclasses, `Error` and `Exception` to distinguish between two semantic branches of issues that can go wrong during program execution.

- The subclasses of `Error` are used to indicate that **something has gone wrong in the JVM itself** in a way that prevents the Java bytecode from being reliably executed any further. The most common reason for this in practice is the JVM running out of stack or heap memory.
- Your program cannot do anything about JVM errors in principle, since nothing guarantees that your program statements will any longer even be executed in any sort of accurate fashion!
- The subclasses of `Exception` are used to indicate that the method cannot proceed in a meaningful fashion. There is nothing wrong with the JVM, it is just that the bytecode that it executes was asked to do something that is logically impossible.
- The language rule to distinguish between checked and unchecked exceptions is that **all subclasses of `RuntimeException` are unchecked**, whereas all other exceptions are checked.

6.7. Assertions

- When testing your code, **it is always better to detect the inevitable failure sooner than later**, so that it is easier for the programmer to trace back the problem to the point that caused the failure, instead of having to slough back through a long diagnostic journey from the observed effect of the bug manifesting itself to its hidden underlying causes.
- Since modern computers can perform hundreds of millions of invisible operations between the bug and the place where the bug finally manifests itself to the user or the programmer, it sure would be nice to be somehow able to detect the bug the moment that it actually happens.
- In testing and debugging code that is under development, it might come handy to seed in **assertions** about the data that is handled by the program in its various points.
- Whenever you believe that some `condition` is logically inevitable at some point of your program, put your proverbial money where your mouth is and `assert` that condition explicitly by writing `assert condition;` to that point in code as a silent tripwire for incorrect logic.
- Whenever the execution reaches the `assert` statement, the `condition` is evaluated. If it is `true`, nothing happens. If it is `false`, an `AssertionError` is automatically thrown to crash your program right on that spot immediately and noisily.
- If your original reasoning was correct, you will never hear about that assertion again. However, if your original reasoning was flawed (or perhaps it was correct, but some later changes to the code made it incorrect), you will find it out the next time the problem actually manifests itself.
- **You should only ever assert things that are 100% under control of your code** so that them being false would logically entail a genuine bug in your code. **You should never assert anything about the outside world** or the method arguments that it passes to you. Your method cannot possibly have any divine powers to dictate how the outside world must behave.
- All this is, of course, also heavily dependent on where you draw the boundary between the "system" and "the outside world". Note also that if some subsystem of your code is intended to be taken out and reused as a library inside different systems, it may not assert anything about the original system inside which it was first developed.
- Assertions should be used only in the development stage and **turned off when delivering the release intended for the end users.** For this reason, the `condition` of an `assert` should never do any actual work needed by the program, such as calling some method that has useful side effects.

- **JUnit** defines its own slew of assertion methods that won't crash the entire program, but these methods throw a custom exception to terminate that test and tell the JUnit's internal bookkeeping mechanism that that particular test somehow failed. (BlueJ shows these assertion failures as black X-marks in the test report window, whereas other exceptions thrown by some methods that crashed during the test show up as red X-marks.)

6.8. Some exception patterns and antipatterns

- The exception handling mechanism in the JVM is not optimized for speed. Furthermore, each exception object builds up and carries the entire stack trace with it, so throwing and handling an exception is a relatively expensive operation.
- **Do not use the exception mechanism as an extralinguistic control structure**, sort of a "return on steroids" that can jump back over multiple levels of method calls in one swoop.
- For the same reason, you should generally not use the famous Pythonic programming idiom of "It is easier to ask forgiveness than permission" in Java code, but rather follow the more conservative principle of "Look before you leap".
- Try to use the existing exception subclasses in the Java standard library whenever possible, as they have standard well-understood semantic messages about the situation that they describe, instead of making up your own exception subclass with the same intended message.
- The number of different exceptions thrown by a method should not depend on the number of different ways that the method can fail, but **on the number of different ways that the caller can fix the problem**. If two different problems are repaired the exact same way, they should be indicated with the exact same type of exception.
- When throwing an exception, throw the most specific exception type that is available to you. Doing otherwise might be termed "passive-aggressive programming" in which the program tells you that something is wrong, but just won't tell you what that was. ("You *should* know.")
- Answering "no" is not a failure that needs to be indicated with an exception, when "no" is the proper and correct answer to the question that was being asked by calling the method.
- Things that will **always inevitably happen during the execution of your program** no matter what (for example, reaching the end of a text file being read line by line by your program) are by definition not exceptional, and thus should not be indicated with exceptions.
- If an exception flies through more than one level of encapsulation, it should be translated to a semantically appropriate type on each level from the second level up, since otherwise that exception will expose the lower level implementation details to the outside world and break encapsulation.

Lecture 7: I/O Streams

7.1. Streams and their decorators

- In Java, the **stream classes** `InputStream` and `OutputStream` model some underlying mechanism to move **raw unsigned bytes** from some point A (**sender**) to some point B (**receiver**). The subclasses of these classes represent actual concrete implementations for different entities that A and B can actually be.
- At the stream level, the transmitted data is a sequence of raw unsigned bytes without any higher level semantics assumed or attached. The universal streaming mechanism transports those bytes from sender to receiver without the slightest care of whatever those bytes might "mean".
- Regardless of the higher level semantics of some data, everything (either code or data) inside a von Neumann architecture computer is nothing but bytes that represent basically whatever you say that they represent. (This is necessary for **metaprogramming** tools such as compilers, interpreters, **transpilers** and **profilers** to be able to exist in the first place.)
- **Every stream in Java is unidirectional** from the sender to the receiver. If you want to perform **bidirectional** communication (for example, over a **TCP/IP internet socket** connection), you need to establish two separate streams to opposite directions.
- If the receiver is busy doing something else when the sender is sending bytes, these bytes won't vanish, as the stream will internally **buffer** the data until the receiver is ready to read them.
- If the receiver tries to read data before the sender has sent it, the `read` method will **block** until either the next byte or the notification of reaching the end of the stream becomes available.
- The methods `read` (in receiver's `InputStream`) and `write` (in sender's `OutputStream`) operate on `int` parameters, since the primitive type `byte` in Java is a **signed byte** with the range -128, … , +127, whereas the raw **unsigned byte** has the range 0, …, +255. An `int` can easily represent and store all these values.
- Either party can `close` the stream at any time, after which no further transmission will be possible through that stream. Once the sender is done writing and closes the stream, **the receiver will receive -1 at the remaining reads** to indicate that no more data is forthcoming.
- A stream built on top of some exclusive resource such as a file or an internet connection should be closed when it is no longer needed, and this `close` should be put in the `finally` block to guarantee that it will happen no matter what happens during the resource allocation and use. (In Java 7 and higher versions, you can use the **try-with-resources** shorthand.)
- **The stream class hierarchy heavily uses decorators to add higher-level capabilities** such as **compression, encryption, data filtering and conversions, and checksum calculation** to existing streams. Each such decorator needs to be written only once, and yet will work with any underlying stream from the same hierarchy, once again vividly illustrating the power of polymorphism and dynamic binding.
- All streams are allowed to internally buffer the data to speed up their work. The method `flush` in `OutputStream` forces the stream to send forward all bytes currently buffered inside it. Calling this method should be **necessary in two-way communication**, such as over an Internet TCP/IP

connection, where you have to wait for an answer to your previous message before you can know what message to send next.

- Flushing might also be necessary during standard output, to ensure that all outputs initiated before the crash get properly written out before the process terminates.
- Closing or flushing the stream decorator will also close or flush the underlying stream, assuming of course that the decorator was properly written to guarantee this.

7.2. Readers and Writers

- For the common situation of transmitting **Unicode textual data** from point A to point B, the class hierarchies of `Reader` and `Writer` work **exactly the same way** as the classes `InputStream` and `OutputStream`, except that the basic unit of data being transmitted is now **one Unicode character** instead of an unsigned byte.
- `Reader` and `Writer` are **facades** to some underlying `InputStream` and `OutputStream` that actually transmits the characters encoded as bytes. (Inside a computer, **everything is made of bytes**, and whatever semantics or aggregation we humans imagine on top of those bytes exists only in our minds as we voluntarily put ourselves on a leash to use some bytes only in some particular way, as opposed to machine code where "Do what thou wilt" shall be the all of law.)
- The facade subclasses `InputStreamReader` and `OutputStreamWriter` that can be used as plugs to convert characters to bytes and vice versa make this relationship explicit, whereas simple classes such as `FileReader` internally create and maintain their own private byte streams.
- One advantage of using these explicit conversion facades between characters and bytes is that you get to choose the encoding scheme. In practice, the UTF-8 encoding is the most popular since it needs only one byte to represent every character in the old **ASCII** range with the codepoints from 0 to +127 (so that any old text file that contains only those characters is already a legal UTF-8 file as it is), which comprise most of the English language text. But the UTF-8 scheme, just like all other standard Unicode encoding schemes, can still encode all Unicode characters.
- Curiously, the methods `read` and `write` of `Reader` and `Writer` are exactly the same as those of `InputStream` and `OutputStream`! This is because the Unicode characters are encoded as integers, and the special value -1 is again used to denote the end of the stream.
- These class hierarchies are still separate, since these methods have different **semantics** despite having the same formal signature. **Semantics exist above the syntax**, so two classes can have a public interface that is syntactically and typewise identical, and yet behave completely differently! For example, consider two classes of `Gunman` and `Artist`, both containing a method `public void draw()`.
- Since it would be rather annoying to write out or read in text only one character at the time, the handy decorators `PrintWriter` and `BufferedReader` are usually placed in front of an existing writer or reader.
- `PrintWriter` offers the powerful and familiar methods `print`, `println` and `printf`, whereas `BufferedReader` offers the method `readLine()` to read the input one line at the time, the special value `null` now indicating the end of data.
- `BufferedReader` allows us to read text data in the spirit of the **Unix command line** so that each data file consists of **records**, each record stored in one line. Each record is broken into **fields**

between the chosen **separator character** that cannot be part of the content of any field. This flexible encoding allows arbitrarily long data fields, and the entire data file can easily be edited by any text editor or further transformed by the plethora of Unix command line tools.

- For more advanced **tokenizing**, **parsing** and error handling of character input emanating from many different kinds of character sources, these days we tend to use the powerful `Scanner` **facade** that can read not only from `Reader`, but from a `String`, directly from a file, or from an arbitrary **character sequence**, again vividly illustrating the power of polymorphic constructors.

7.3. Dependencies between classes

- The **Open-Closed Principle** states that classes should be **open for extension** (inheritance) but **closed for modification**. Once a class has been successfully compiled, in principle **its source code should never be needed again in writing other classes or subtypes of that same class**.
- One should generally avoid **reuse by copy-paste**, the very lowest form of software engineering. In fact, the more you copy-paste your own code within the same project, the more you should feel sorrow for being in the state of sin and in need of serious purification and repentance by means of **refactoring** your classes and methods.
- Besides, combining similar methods and blocks of code into their most general parametric form usually helps you understand the underlying phenomenon better, which allows you to improve your logic in ways that would have been impossible with the original implementation. All kinds of modifications become much easier whenever every truth about the program is expressed in exactly one place in the code.
- **There should never be a need to recompile a class**, except when the internal implementation of that class changes to an updated better version, or new functionality is added to the class in the next version of the entire library.
- Class `A` **depends on** class `B`, if editing and compiling class `B` makes it necessary to recompile also class `A`. Therefore, to keep the classes properly closed for modification, **each class should only depend on classes that are less likely to change**. Otherwise, changing one class would trigger the recompilation of all other classes that depend on that class, written by everyone in the world!
- Counterintuitively, **the higher level abstractions are more solid and less likely to change than concrete classes**, opposite to how we normally think about concepts of solidity and concreteness. Therefore, as instructed by the **Dependency Inversion** principle, **all classes should depend on high level abstractions, and never depend on any concrete low level implementations**. This is no different from how in the physical world, we prefer to build houses on solid rock instead of quicksand.
- In general, try to make the parameter types of your polymorphic method as high level and general as possible. For example, if you write a method that expects an `ArrayList<E>` as its parameter, ask yourself if this parameter really needs to be specifically an `ArrayList<E>`, or whether it could be any `List<E>`, or even any `Collection<E>`.
- (However, a method that could handle any `Object` in a meaningful fashion is probably somehow wrong in other ways. Outside metaprogramming, such methods should not really exist in modern Java. Methods that can accept and return arbitrary objects indicate the need to turn that class into a **generic** version of that method.)

- Once you hardwire your class to depend on low level implementation decisions made in some other class, any future change in such decisions will force you to rewrite your class. **This is why we also use encapsulation to make all implementation details `private` so that other classes can't possibly get attached to them**, giving us the freedom to change these details later.
- Once the `public` interface of some class is used by enough people, that interface must be considered to be set in stone so that it may never again change in ways that are not backwards compatible. The `public` interface of any class that is intended to be used by others therefore must be very carefully designed, since it has to be correct the first time.
- (In designing the public interface of an abstract superclass, you should always also create some concrete subclasses of it to ensure that all aspects of that interface can be realized in practice. You should also write some polymorphic methods whose parameter type is the abstract superclass, to ensure that the interface is complete so that it is possible to do things with it without any extra information about the actual subtype.)
- Implementations of methods can always be changed and optimized later, but changing the public interface would break all subclasses that have been written to implement it.
- Since Java 8, any interface can always be expanded with new `default` methods without breaking any existing code that depends on that interface.

7.4. Reflection

- The **dynamic binding** mechanism of Java and similar languages allows you to call the method `foo` with `x.fly();` without knowing or caring about the exact subtype of the object `x`. However, the superclass that defines the method `fly` must be available during the compilation of your code.
- The `instanceof` operator allows you to check whether some object is an instance of some class that is known to you at the compile time. You cannot use this operator to check the type of an object against a class that is determined at runtime.
- Various **metaprogramming tools** such as **BlueJ** or **JUnit** need to have a much stronger ability to crack open and inspect any arbitrary object and determine its type and contents, even if this type comes from some class hierarchy that did not exist at the time that the tool itself was compiled.
- In the Java type system, **all classes are also objects**, instances of the **metaclass** called `Class`. (Some wag should turn this into an Abbott and Costello style "Who's on first?" rapid talking comedy act.) To prevent an infinite chain of turtles all the way down, the class `Class` is an instance of itself, having given birth to itself at the time when the Java language itself sprung to existence.
- **An object is a class if it allows new instances of itself to be created**. For example, `String` is a class, since the string object `"Hello, world!"` can be created from it. However, this object is not a class, since no object can possibly be an instance of it.
- Objects can be cloned with the `clone` method, but that creates another identical instance of the same type. Cloning requires that the class implements the `Cloneable` marker interface.
- Every time the Java virtual machine loads up the bytecode of some class, it automatically creates **the class object** to represent that class, filling it with the information that it knows about what that class contains.

- To dynamically acquire the class object of an arbitrary object `x`, call `x.getClass()`. The method `getClass` is defined to be `native final` in the universal superclass `Object`.
- To use the class object for the class `Foo` known at compile time, use special syntax `Foo.class`.
- The class object can be asked what fields, methods, constructors and nested classes it contains, using the **wide reflection** classes `Field`, `Method`, and `Constructor`. These classes have exactly the methods that a reasonable person would expect. You can essentially crack open any object or class given to you and find out everything about everything that it contains.
- Note that `private` is only a **safety** feature in Java programming, but has no **security** powers of any kind, since it can always be bypassed with clever use of reflection.
- The field, method and constructor objects even allow fields and methods of an arbitrary object to be accessed and modified dynamically, even if those members are `private`. However, this is rather slow, and should only be used in metaprogramming. If you know the type of some object, just access its fields and methods directly and let the compiler enforce your type safety for you.

7.5. Serialization of arbitrary object structures

- **Serialization** is a powerful mechanism in Java that uses reflection **to convert arbitrary objects into a series of raw bytes** that can be transmitted along streams. These objects can be stored in files and databases for **persistence**, meaning that the object outlives the process that created it.
- The facades `ObjectOutputStream` (with method `writeObject`) and `ObjectInputStream` (with methods `readObject`) can convert and transmit not just arbitrary objects, but arbitrarily large **object structures**.
- In Java 11, **serialization was officially deprecated as a security hole.**
- In general, when you need to store data objects into a file or some other form of persistent storage, they should be encoded into some **language-independent standard representation** such as **XML** or **JSON**. These representations use **raw text as universal data encoding** and can therefore be meaningfully read in all languages (unlike Java object serialization binary format that works only in Java) and can easily be further processed with various tools or simple text scripts.
- Serializing an object will automatically follow all references emanating from that object, and **recursively serialize all objects hanging from it**, creating a **deep copy** of the entire structure. An entire collection with all the objects stored inside it can be serialized with one method call!
- (Serializing an object that was constructed from a non-static nested class will therefore also serialize its outer class context object, and then every object reachable from that. Be careful!)
- The object stream is even smart enough to remember which objects it has already serialized, and if the same object is serialized again in the same stream, the serialization mechanism instead encodes a message for the receiving end to reuse the previously received object.
- For this reason, the serialization does not get into an infinite loop even if the object structure contains **cycles of references**, that infamous bane of all programming with pointers.
- The only restriction for serialization is that the objects involved all must implement the **marker interface** `java.io.Serializable` to allow it. If the serialization is given an object that is not an `instanceof` this marker interface, the serialization fails and throws an exception.

- Some objects are so inherently tied to the particular Java virtual machine process and the underlying platform that it would make zero sense to save them to files for future access, or transport them over the Internet to a completely different machine.
- To force the serialization to ignore a reference field that points to an object that is not `Serializable`, you can define that reference to be `transient`. (In the receiving end, that reference field in the copy of the object will be `null`.)
- Serialization only serializes the object data, but not the executable class bytecode of the class the object was constructed from. (Sure, the ability to transmit executable code would have been awesomely cool among consenting adults, but also an even more horrendous security hole.)
- To protect against the possibility of the receiving end having a different version of the class and its bytecode, the serialization also attaches a **64-bit checksum of the class bytecode** into each object that is serialized.
- Perhaps a bit surprisingly, all enums are `Serializable` in Java. Of course the JVM is smart enough to enforce the uniqueness of each object from that enum within that JVM process.

Lecture 8: Generics And Other Nifty Features

8.1. Minor language improvements back in Java 5

- During the compilation of a Java class, the compiler automatically sees all the classes in the current package (which is the **default package** in the absence of a package declaration) and the package `java.lang`.
- Ordinary `import` makes one class or all classes in some package **visible to the compiler** so that you don't need to write their fully qualified names every time you use those classes. A `static import` makes one or all `public static` members of some class visible to the compiler so that you can use them directly without having to prefix their names with the name of that class.
- This only saves a bit of typing, but of course `sin(x)` looks less clunky than `Math.sin(x)`, and `sort(arr)` looks better (at least more Pythonic) than `Arrays.sort(arr)`.
- We have already seen how to use method overloading to create versions of methods for different types of parameters. However, **variable length arguments** (or **varargs**) allow some method to handle any number of arguments not known at the time the method itself is written.
- A method can be defined to take an arbitrary number (that is, **zero or more**) **varargs** of a given type. This vararg parameter **must always be the last parameter in the method parameter list**, and is denoted syntactically with an **ellipsis** (...) after the parameter type.
- This mechanism works pretty much the same way as `*args` in Python. Since Java does not have keyword arguments, it has no analogue of `**kwargs` in Python.
- (Oddly enough, Python 3 now also has an actual **ellipsis operator** in the language as a built-in constant symbol that currently does nothing in the core language, but can be handy **syntactic sugar** for various current and future extensions such as **numpy**.)
- Inside the method body, the varargs are automatically collected into an array of the same name. This array can even be empty, as it should be when the caller has no varargs to give. (In programming, **zero is always a possibility** that our code must be able to handle.)

- The varargs can also be given as an array, which allows one vararg method to call another vararg method, passing its arguments to the second method, no matter how many were given.
- In the Java standard library, vararg methods are perhaps most often seen as **factory methods**. The canonical name for such methods is `of`.
- Perhaps the two most commonly used vararg methods in the present Java standard library are `printf` in `PrintWriter` and `asList` in `Arrays`.

8.2. Autoboxing

- For every primitive type, Java standard library defines a **wrapper class** to represent **a value of that type as a proper object**. For example, `Integer` for `int`, `Double` for `double`, etc.
- Wrapper classes also have handy `static` methods and constants for common operations with values of that particular type, such as `Character.isWhitespace` or `Double.isInfinite`.
- Especially since the Unicode standard defines tens of thousands of characters with a multitude of properties, you should always use the utility methods of `Character` to check for these properties to guarantee full portability and internationalization. Never try to write these methods yourself.
- Before Java 5, the conversion back and forth between the primitive and the wrapper had to be always done explicitly. From Java 5 onwards, the compiler silently inserts this conversion when needed when the compilation would otherwise fail due to a type error of using a primitive when a wrapper was expected and vice versa, so the primitive and the wrapper types can now be used **almost** interchangeably… if it weren't for a couple of nasty little pitfalls!
- **Wrapper classes are immutable**, and therefore very inefficient when you need to iterate through a large range of values one at the time. If you use a primitive type, the same small group of bytes are always reused when moving to the next value, whereas using the wrapper class necessitates the creation of a new object each time to represent the current value.
- (Every immutable type such as `String` has this exact same problem, which is why we don't make all data immutable, unlike those bearded weirdos in the **pure functional programming** school of thought. If your complicated data structure really needs to be immutable, **a linked list with insert and remove from front** is the best structure. **Trees** are also good, assuming that you can share identical branches.)
- Equality comparison `x == y` does not perform the intended value comparison when `x` and `y` are both wrapper objects. The more fundamental language rule of Java that **the operator == always compares the memory addresses of the objects** takes precedence over autoboxing.
- To make this whole issue even more confusing, the class `Integer` **internally caches** the boxed versions of values in the range from -128 to +127. This works out swimmingly when you, for example, create a `Map<Something, Integer>` to keep count of how many times you have seen different values of `Something`, since up to 127 possible `Integer` objects are shared through this entire map, creating a new `Integer` object only when some count reaches 128.
- (This is a good example of the **flyweight** design pattern, where identical immutable parts of a large collection of objects are shared instead of creating redundant copies. The canonical example of this pattern would be a **rich text editor** with font and other info potentially per character, but most of the time identical for all characters, unless you are writing some sort of ransom note.)
- In comparing `x == y` where only one variable is a primitive, the wrapper is sensibly unboxed.

- A primitive field `boolean x` will be automatically initialized to `false`, whereas a reference field `Boolean y` will be initialized to `null`. Unboxing `null` will fail and throw an exception.

8.3. Annotations

- By definition, **a comment cannot affect the compilation or runtime behaviour of the code**, since the **compiler always skips every comment** without looking at its contents. (More specifically, each comment is replaced by a single whitespace character.)
- **The JavaDoc tool** is the opposite in that it skips the code and reads the specially marked comments to generate HTML documentation pages.
- **Annotations** are kind of "smart comments" that can be attached to classes, fields and methods, and even to other annotations in the case of **meta-annotations**.
- Annotations exist during both compilation and runtime, and unlike comments, can affect both the compilation and the program runtime behaviour (the latter via **reflection** and `getAnnotations` method).
- Annotations are themselves types, and in addition to **the standard annotations** defined in the language, you can easily define your own annotation types with `@interface`.
- Syntactically, annotation types are distinguished from classes by prefixing their name with `@`, the name "at" of this symbol coincidentally being an abbreviation of "annotation type".
- In practice, `@Override` is surely the most commonly used annotation. It can be applied only to methods, and it causes compilation to fail if the annotated method does not actually override some superclass method. Such a mishap typically happens because of a typo in the method name, or when you accidentally use a more specific parameter type in the parameter list, and therefore accidentally **overload** the method instead of overriding it as intended.
- Metaprogramming tools such as **JUnit** use class and method annotations to decide what to do with them. For example, JUnit recognizes the tester methods and the tester class from their `@Test` annotation. (Before annotations, each tester method name had to begin with the word "test" for it to be recognized by JUnit reflection mechanism, which made automated testing clunky and prone to typos.)
- Even an ordinary program can use reflection and class loading to load up new classes on the fly and use their annotations to determine what to do with these classes.
- To combat the famous "billion dollar mistake of software engineering" of `null` references, some tools associated with Java 8 introduce a handy **type annotation** `@NonNull` that allows the compiler to verify that a variable thus annotated cannot possibly be `null`. Such a variable must be initialized with something that is also `@NonNull`, and the same requirement holds for every assignment afterwards.
- (This is similar to the distinction between the guaranteed non-null and possibly-null types `Foo` and `Foo?` in the more modern **Swift language**, although more annoyingly verbose.)
- Various other annotations have been proposed more or less seriously.
- Java 8 also introduced the generic type `Optional<T>` that is a wrapper that either contains an object of type `T`, or denotes the absence of such an object in a controlled fashion. A modern method whose return type is `T` should never return `null`, but return an `Optional<T>`.

8.4. Basic generics

- So far, the rule for static type checking of Java has been that **all data that the program talks about must be given an explicit type at the compile time**. The compiler enforces that data is used only in ways that are meaningful for that particular type. This prevents programs from crashing at runtime because of type violations.
- The downside of this type safety is that since we don't want to repeat ourselves, the Collection Framework methods must operate on `Object` parameters and return values, which then necessitate result downcasts that by their nature break down the type safety.
- Generics relax this requirement so that in a generic class, some types used in the code (for example, method return type or the type of a field) are written using **placeholder names** "to be filled in later" at the time when the class is actually used.
- The placeholder names are given as **type parameters** in angle brackets after the class name, conventionally using a single capital letter, such as `ArrayList<E>`, to distinguish them from the names of actual classes. (No actual class should ever have a single-character name, so you can always assume a single-character name to be such a placeholder when reading through code.)
- The compiler will still enforce that inside that class, these unknown types are used in a consistent and legal manner based on what little is known about them. For example, we don't need to know what actual type `E` is instantiated with to know that it surely must be legal to assign from one variable of that type into another variable of that same type.
- The user code can then apply **type arguments** to create as many different **type instantiations** of the generic type as are needed. Inside the type system, the two different instantiations `ArrayList<String>` and `ArrayList<Integer>` of the same generic class are two separate types that are not assignment-compatible with each other.
- In the instantiation `ArrayList<String>`, the method `get` has the return type of `String`, so **no downcast is needed** when assigning the result of this method to a `String` variable. In the instantiation `ArrayList<Integer>`, the otherwise exact same method would have the return type `Integer`.
- With the **diamond operator** `ArrayList<>`, the compiler fills in the type argument from the surrounding context. This saves a lot of annoying typing if the type argument is itself a complicated generic type, something like `ArrayList<List<Set<String>>>` for an arraylist whose each element is a list of sets of strings.
- Of the four possible combinations generic-generic, generic-nongeneric, nongeneric-generic and nongeneric-nongeneric, **all four are good and meaningful for a superclass and its subclasses**.
- **Enums, throwables and anonymous inner classes** cannot be made generic at all.
- An individual **method can also be made generic**, independent of the generics of its surrounding class. When you call a generic method, **the compiler can in most situations infer its type arguments**, so you don't need to give them in the call but simply call the method the normal way.

8.5. Erasure and its consequences

- Java generics very much resemble **C++ templates**. In practice, most of the time in practice there is no real difference in their use in both languages.
- However, an important difference between Java and C++ is that in C++, the compiler produces a separately compiled version for each type instantiation of the generic type. This can result in **code bloat** in the generated binary executable, but on the other hand, allows the compiler to customize and optimize each separate type instantiation to the maximum extent.
- In Java, the **distinction between different type instantiations exists only at the compile time**. Once the compile-time type checking is complete, **all generics are erased** so that the separate instantiations actually become one and the same **raw type** produced by the compiler.
- The **raw type** is the generic type without any type arguments.
- Since the code was verified during compilation to obey the "invisible fences" of each type declaration, removing these invisible fences does not change the fact that this code will still obey those very same fences at the runtime! Dynamic binding of Java method calls based on the object's runtime type makes each separate type instantiation use its correct subclass method at the runtime even after all the information about the different type arguments has vanished.
- (**Seriously**, think carefully about that previous item. **Its message is very important**. For the exact same reason, the primitive types don't need to actually exist in the runtime once the compiler has produced the correct machine code instructions to operate on such variables.)
- Even if you instantiate the same generic type with a hundred different type arguments in your program, the size of the resulting bytecode file remains unaffected.
- However, because of the way erasure works, a generic class `Foo<T>` cannot (1) use the type parameter `T` in any `static` member in any role, or (2) use `new T()` (besides, nothing guarantees that `T` has a `public` no-argument constructor), or (3) use `x instanceof T`. At the runtime when the type `T` no longer exists, what could these runtime operations possibly do?
- Method overloading cannot depend on generic type parameters, since nothing guarantees that those parameters are instantiated in a way different from the other overloaded versions.
- Generic classes can have `static` fields as long as their type is independent of `T`, but one copy of that `static` field is shared by all instantiations of the same generic class.

8.6. More on raw types and reification

- For every generic type, the corresponding **raw type** exists not only after compilation, but during it in the type system, since it is needed for interaction with all legacy code that predates generics. This is unfortunate but necessary, unless we could reboot the language from scratch.
- (Many things in Java would be different if the language were redesigned from scratch by dropping the backwards compatibility. This was done with the creation of **Scala** and **Kotlin** as bytecode-compatible redesigns of Java to make the language more suited for this millennium.)
- Types that exist at runtime in Java are said to be **reifiable**.
- However, **raw types should never be used in new code**, and the compiler issues the "unchecked" warning if you do that, since **raw types break all type safety**. Anything goes!

- In the type system, **raw type can be assigned back and forth between any type instantiation of that generic type**. This allows us to easily create, say, an `ArrayList<Bird>` reference pointing to an object whose type is `ArrayList<BankAccount>`.
- To avoid the impossible situation where a bank account is asked to fly at runtime, **the Java compiler adds a silent downcast (and the corresponding runtime check)** to every return value assignment of a generic method. So the downcasts that we had to write explicitly before generics are still there but hidden in the bytecode, wasting processor cycles.
- Method `Collections.checkedCollection` returns a decorator that checks the type of the added object at the time of addition, instead of at possibly much later time when that element is accessed with an unsuccessful downcast.
- In Java, **you cannot create a primitive array whose element type is any generic type**, even if that generic type is fully instantiated. In Java heap, each array object must know its element type, but obviously this would be impossible when the element type does not exist at runtime!
- These days we can always write a generic method `void <T> foo(T[] arr)` for that purpose.

8.7. Inheritance relationships over generic types

- Keeping the eye carefully on the prize, remember that the purpose of inheritance was to allow us to write polymorphic methods that can operate on all subtypes of some abstract supertype, instead of having to write a separate version of that method for each of those subtypes. We sure don't want to give up this powerful ability to pay for the possibility to have generic types.
- At first sight, it is not obvious that we would even have to give up anything. Just like we can have a polymorphic method `void foo(Bird b)` that can be given any `Hawk` or a `Sparrow` instance as an argument, could we not write a method `void foo(Collection<Bird> cb)` that could operate on any collection of any kinds of birds, so that this method could be given an `ArrayList<Albatross>` or a `HashSet<Hawk>` instance as an argument?
- The method itself compiles just fine, but calling it using an argument of either of the previous types fails to compile with an error message about incompatible types. What is going on here?
- **Liskov Substitution Principle** reveals the highly counterintuitive reason for this paradox: even though `Hawk` is a subclass of `Bird`, and `ArrayList` is a subclass of `Collection`, their generic combination `ArrayList<Hawk>` is not a subclass of `Collection<Bird>` !
- You can add an instance of `Sparrow` into a collection of birds, but you can't add one into an arraylist of hawks. **Therefore the latter collection cannot possibly be a subclass of the former**, since LSP requires that the subclass must be able to do everything that the superclass promises to be able to do!
- `ArrayList<Bird>` is a subclass of `Collection<Bird>`. No problem there, as long as the generic type arguments are identical.
- The problem is that `Collection<Bird>` is a collection that is able to contain all kinds of birds, whereas `ArrayList<Hawk>` is not. These two are separate types, and trying to pass one as the other is no less absurd than trying to pass, say, a `BankAccount` instance as a `Sparrow`.
- Because arrays predate generics, they were designed to be **covariant**, meaning that `Object[]` is supertype of `Bird[]`. This makes zero sense and really, really, really ought to not be this way,

but it was made so to allow writing polymorphic methods that can be given any array object as argument, instead of having to define a new version of that method for every possible array type.

8.8. Bounded types

- To implement the previous polymorphic method as we intended, we need a type that means "A homogeneous collection of some particular but presently unknown kind of a bird to be decided later," instead of "A heterogeneous collection that can contain all kinds of birds simultaneously", which `Collection<Bird>` is.
- **Bounded types** allow us to define exactly such types, with either a **wildcard** question mark or a type parameter name bounded either above (with `extends`) or below (with `super`) to restrict which types are possible as instantiations of that type parameter.
- `Collection<? extends Bird>` is a collection of some particular subtype of `Bird` that we don't know at the time other than that it will be exactly one particular type. (Perhaps that type will be an owl, but "who" knows?)
- A polymorphic method whose argument is such bounded parameter type can be given an arraylist of albatrosses, a hashset of hawks, or any particular instantiation that anyone could ever think of. The method call then no longer causes a compile time type error.
- The bounded type provides enough information about the unknown type so that the compiler can allow calling the methods of `Bird` to the elements of the collection given to it as argument. Dynamic binding, once again, ensures that the correct subclass version of that method is executed at the time of the call.
- Unlike with a parameter of type `Collection<Bird>`, the polymorphic method cannot `add` an object of type `Owl` to the `Collection<? extends Bird>`, because nothing guarantees that this unknown subtype would specifically be an owl. It might be sparrow or eagle, or whatever.
- To allow such addition, use the lower bound type `Collection<? super Owl>` instead.
- All bounded types behave as abstract classes in that they **can only be used as types of variables and method parameters**, but they cannot be used to create actual objects.
- However, even though you can't create an object of type `ArrayList<?>`, you **can** create an object of, say, type `ArrayList<List<?>>`. The element type of this arraylist **is fully known**: a homogeneous list of some unknown element type. (Yeah, read this item again a few times.)
- A type bound can even be recursive, such as `List<T extends Comparable<T>>` for lists whose element type is something that is order comparable with itself.
- Hardcore Java and generics enthusiasts can try their hand in deciphering the intention behind the doubly recursive type `Enum<E extends Enum<E>>`.

	`ArrayList`	`ArrayList<?>`	`ArrayList<Object>`
Can create objects of using `new`	Y	N	Y
Supertype of `ArrayList<String>`	Y	Y	N
Exists at runtime	Y	N	N

Guarantees runtime type safety	N	Y	Y

Lecture 9: Concurrency Fundamentals

9.1. Threads and concurrent execution in JVM

- It is useful to distinguish between the two terms **concurrency** and **parallelism**, the former meaning that **multiple logical computations are being executed partially interleaved** (as opposed to **sequentially** waiting for one computation to end before the next one is started), and the latter meaning that these multiple computations are **simultaneous in the physical sense**.
- All parallelism is concurrent, but not necessarily the other way around.
- The **Java Virtual Machine** is one of the **processes** running inside your computer. The execution of JVM simulates an imaginary computer architecture that executes **bytecode instructions**, this imaginary computer safely residing in a **virtual sandbox** inside your real computer.
- Of course, theoretically this "real" computer could be just another virtual machine, with no theoretical upper limit how many levels of virtual machines could be piled up on top of the actual physical computer that is run by the physical universe... which may or may not itself be a computational simulation running inside some higher level universe. You can be certain of one thing that you exist, but can never truly know which level of The Matrix you exist in.
- The data memory inside the JVM is divided into the shared **bytecode** and **object heap** areas shared by all threads, and **local variable stack areas** separately for each thread, jealously guarded against improper use by the JVM.
- The JVM loads up the class bytecode files in the bytecode area as needed in the program. There are no **introspective** bytecode instructions that could access or modify the contents of this bytecode area. It is important for both safety and security that a Java program cannot modify itself during its execution, except by expanding itself by loading more classes in a controlled fashion through the JVM services.
- The execution of the bytecode program is divided into separate threads. Each thread has a **program counter** that points to the position in the bytecode that the execution of that thread is currently at.
- Each thread has its own stack of local variables. This is necessary because in general, different threads are executing different parts of the program.
- **All threads share one and the same object heap** so that the communication between Java threads takes place by **reading and writing shared objects**.
- Concurrency using **shared memory** under the same physical hardware makes concurrency simpler than general **distributed computation** where the concurrent entities are executed in separate computers connected in a **network** that introduces an unpredictable delay to **message passing** between the concurrent entities, and in which even something as elementary as knowing the exact number of participants might not be available to any particular entity.

9.2. Java execution model

- The JVM starts up by creating the **main thread** that starts its execution from the main method of the program, and the **garbage collector daemon thread** that looks for unreachable objects in the heap and releases the memory used to store them.
- **When only daemon threads remain, the JVM process terminates**. Unless you have created additional threads yourself, this effectively means that the JVM terminates once the program's `main` method returns, which is exactly what you are accustomed to so far.
- New threads can be dynamically launched during the execution of the program. This should not be done willy-nilly in real programs, since launching a new thread is a relatively expensive operation, as it requires allocation of the **thread local stack** from the memory available to the JVM process, for example.
- In the classic single-processor system, the JVM keeps one of the threads active and executes it by repeatedly reading in the next bytecode instruction and simulating its effect to the stack and the object heap. The execution then automatically continues from the next instruction, unless the current instruction was some kind of (either conditional or unconditional) jump.
- At any time, the JVM is free to **context switch** to move on to execute any other thread. **The threads and the Java program code that they execute have no control whatsoever over this**. The context switches can happen completely unpredictably at any time, and the threads are not guaranteed to get **execution time slices** in any kind of **round robin** fashion.
- For our slow human eyes and brains, it looks like all these threads are truly executed in parallel, since just like watching a movie at ~~25~~ 60 fps, our slow human eyes and brains can't tell apart the individual execution time slices.
- Such **preemptive multitasking** works much better than the more primitive **co-operative multitasking** where each thread always has to explicitly give up its turn, which would allow some malfunctioning thread stuck in an infinite loop to also freeze all other innocent threads.
- Some programming languages allow threads to be broken into finer co-operative **coroutines** or **fibers** that operate on a co-operative multitasking discipline, so that each coroutine has to explicitly give up its turn. Good use for this technique are various **producer-consumer systems**. (In modern Python, such situations are often best handled with **generators** that `yield` elements of some virtual data stream one element at the time, passing the control back and forth between the generator object and the object that consumes these elements.)
- In the computer operating system, different **processes** operate the same way one level higher in a preemptive multitasking model. The semantic difference between "threads" and "processes" is that **threads share memory**, whereas each process runs in its own separate memory protected from other processes, and is thus forced to do its communication with other processes through the dedicated operating system services.

9.3. Threads in Java language

- It is extremely easy to launch a new thread in Java: simply create a new object from `Thread` or any custom subclass of `Thread`. This class is special in that the threads inside the JVM process always correspond one-to-one to the instances of `Thread`.
- It would be rather annoying if every new thread started executing from the `main` method. Instead, each thread is given a `Runnable` object as a constructor parameter whose `run` method it will then execute. You simply implement this `run` method to do what you want the thread to do.
- To actually make the thread to execute the `run` method, you need to call its method `start`. Once started, the **daemon status** and **priority** of that thread can no longer be changed.
- From the `run` method, the execution can go to whatever other parts of the program that it can reach through the method calls. Once the execution returns from the top level call of `run`, the thread terminates, and cannot be restarted.
- Launching and managing threads yourself is a bit too low level thinking for high level object oriented programming. Instead, you rather create one instance of `ExecutorService`, a handy utility that juggles threads behind the scenes so that you don't have to.
- You can `submit` any number of `Runnable` tasks to the same `ExecutorService` instance. They get scheduled and executed in some **asynchronous** order by the threads privately held by the service.
- When you submit a task, you immediately get back a `Future<T>` object. This object acts as a **receipt** that you can later use to inquire about the status of your task, cancel the task, etc. Unlike an ordinary return value of type `T` that contains the result of a completed computation, the `Future<T>` object represents the result of a future computation that is yet to complete.
- The interface `Runnable` and its method `run` are rather primitive, since you can't return a result from a `void` method, nor can you override this method to throw any checked exceptions, since the superclass version didn't declare any. A more modern interface `Callable<T>` with its much better designed method `T call() throws Exception` can be used in place of `Runnable` in submitting tasks to an `ExecutorService`.
- After you submit a `Callable<T>` task, the `Future<T>` receipt can also be used to query the exception thrown by the failed task, which is not possible with `Runnable`.

9.4 Critical sections and race conditions

- A computational operation is said to be **atomic** if no context switches can happen during it. The outside world cannot observe any intermediate stage of data inside an atomic operation.
- Very few operations in Java are truly atomic, such as `int` assignment.
- Even an assignment from an eight-byte `long` to another `long` might happen only halfway through, with a context switch taking place in the middle so that the `long` contains four bytes from the source and four bytes of the original value!
- When two threads access and modify the same object with **non-atomic operations** that could be interrupted in the middle by an unlucky context switch, the resulting execution interleaving can cause the object and the surrounding computation to be left in an inconsistent state.

- (The related term **failure atomicity** means that if an exception is thrown inside a non-atomic operation, the object returns to the state it was before that operation began. A failure atomic operation therefore happens either completely or not at all. Failure atomicity is rather important in **distributed databases** that contain, for example, banking information.)
- **Immutable objects** are always **thread safe** since they cannot be modified, and thus need no synchronization regardless of how many threads access them concurrently. (Redundant identical copies of immutable objects might occasionally get created, but this affects only the program memory efficiency, not its correctness assuming that `equals` method does not use `==`, and these days we will cheerfully trade memory for guaranteed good time and correctness.)
- The combined effect of two unfortunately interleaved threads **typically manifests as lost or missing data or values**. Such **concurrency bugs** resulting from **race conditions** can be extremely difficult to find and debug.
- Java Collections and **especially I/O streams** are **not thread-safe** (except for some particular implementations that explicitly guarantee this by their design), so that they have no built-in protection against race conditions and can thus behave strangely in concurrency.
- First of all, it is not possible to make some operation atomic by telling JVM not to make context switches during it, as the threads have no control over context switches. (Even if such a global ban on context switching were possible, forcing it to happen would basically eradicate all benefits of parallel execution using multiple processor cores.)
- Since the JVM executes bytecode instructions instead of Java statements, which are broken down into a series of bytecode instructions that can then be arbitrarily combined and rearranged by various compiler optimizations, a context switch could very well happen "inside" some statement! **You therefore cannot magically make some complex operation atomic by writing it as a clever one-liner.**
- To solve this problem, you need to identify from your code all **critical sections**, blocks of code that should be executed by at most one thread at the same time so that while one thread is executing the critical section, no other thread may enter it.
- (Technically, we could allow two threads to enter the same critical section provided that they are operating on separate objects. However, the tiny efficiency gains available from such much more fine-grained locking are not usually worth the effort of complicating your code.)
- **Only you can identify what constitutes a critical section**, since they depend on your intended semantics of the program. No compiler can identify critical sections for you, even in principle.
- Even worse, the one and the same critical section is not necessarily one continuous code block inside the program, but can exist in multiple pieces in separate methods, even separate classes!

9.5. Mutual exclusion locks

- Having identified a critical section, you should declare a **lock object** of a suitable subtype of the interface `java.util.concurrent.Lock` (typically `ReentrantLock`), this lock being shared by all threads that will be accessing the critical section. This lock object would be best named `mutex` or something similar to instantly document its purpose.
- A lock can be **owned** by **at most one thread at any time**. A thread can **capture the lock** by calling its method `lock`, and released by calling `unlock`. If another thread (or "threat", as one

student made an unintentionally amazing misspelling in the final exam) tries to capture a lock that is owned by another thread, this method `lock` will **block** until the time that the lock becomes available. The JVM will simply context switch to execute some other thread, since it would be pointless to execute the thread that is waiting for a lock owned by someone else.

- In the beginning of each critical section, call `mutex.lock();` and at the end of that critical section, call `mutex.unlock();`
- (If the execution of this critical section might crash and throw an exception, make sure that you release the lock in the `finally` block, to ensure that the lock does not remain forever in the locked state so that nobody can ever again get in.)
- The mutex lock effectively acts as some kind of invisible force field (or to use a more mundane analogy, an ordinary bathroom privacy lock) that prevents other threads from entering the critical section while the execution of some other thread is inside it. Once the lock is released on the way out, the force field vanishes, so that the next thread can get in, locking the critical section again for the duration of its execution.
- If the critical section is split into multiple disjoint pieces in the source code, **each piece must use the same mutex lock**. Two separate critical sections that are independent of each other can use separate locks to avoid needlessly blocking each other.
- The Collection Framework classes are typically not synchronized against concurrent access. The static method `Collections.synchronizedCollection` (similarly for `List` and `Set`) decorates any underlying collection with a decorator object whose three dynamic set methods `add`, `contains` and `remove` are internally synchronized within this method. (Iteration cannot, and should not, be synchronized this way.)
- The package `java.util.concurrent` offers several list and set implementations specifically designed to tolerate and even thrive under concurrency.
- **A reentrant lock** can be **owned by the same thread multiple times**, which makes a difference if the critical section is **recursive**. The lock maintains an internal counter of how many times its current owner has captured it, and the owner must then release it that many times for that lock to become available for other threads.
- (In computer programming, a system is said to be **reentrant** if multiple activations of that system can exist simultaneously. This forbids storing data used by that system in global or static variables, of which **there can always be exactly one**. As a rule, you should never use global variables anyway, except as `static final` named constants.)
- When designing synchronization, it is better to lock a little bit too much (so that possibly once in a blue moon some thread is forced to wait a little while when it really wouldn't need to) than to not synchronize enough and by doing so break the correctness of the entire program! However, forcing too much synchronization on a program may cause it to **deadlock**.
- Nobody ever said that concurrent programming was going to be easy: it is a whole another dimension extending away from ordinary sequential programming.

Lecture 10: Concurrency Controls

10.1. Condition variables

- Mutual exclusion is just one of the two types of synchronization needed in concurrent programs. The ability to pause a thread until some condition becomes true is just as important, and very much needed in writing methods that **block** until it becomes possible for them to proceed.
- For example, a method that needs some exclusive resources such as files has to wait until that resource becomes available, if the resource is currently being held by another thread or process.
- **Whether some method is blocking must be documented for the users**, since every time someone calls a blocking method, they take the risk of their thread freezing for an unpredictably long time.
- Some libraries can offer alternative **timeout** or **fail-fast versions** of those methods that throw an exception instead of blocking, and so let the caller decide how to proceed from there.
- Blocking methods, as a rule, throw `InterruptedException` in their signature (see below).
- **Blocking is infectious** in the sense that if your method calls any blocking method, then your method is also blocking, from the point of view of its callers.
- A **busy wait** (also called a **spinwait**) that uses an empty loop `while(!CONDITION) {}` is a **horrendously inefficient** and bone-headed way to implement blocking on a condition.
- **Condition variables** (in older literature, also called **monitors** when paired with a mutual exclusion lock, as the Java `Condition` objects always are) are a simpler and much better way to implement waiting so that a thread that is waiting for some condition does not waste any processor cycles in running around in circles like a dog chasing its own tail.
- When you identify a condition from your program that some thread is waiting, create an instance of `Condition` to represent that condition. This instance must be a field that is visible to both the waiting thread and to whoever will make the condition to become true.
- Modify the previous busy wait loop so that instead of having an empty body to be `while(!CONDITION) { condition.await(); }`, it calls the method `await` of the condition variable object. The JVM will not waste any time executing any thread that is known to be in a waiting state.
- It is your responsibility to identify every single part of your program that could theoretically make that condition true for some awaiting thread. (The compiler cannot help you in this, even in principle.) In all those places, you should call the method `signal` of that condition variable to wake up one waiting thread.

10.2. Crowding inside a condition variable

- **False alarms** that `signal` some waiting thread to wake up cause no problem in principle, since the condition of the while-loop that surrounds the `await` will merely cause that thread to go back to the waiting state again. No harm, no foul, no problemo, as my fellow kids like to say to sound hip and relevant to these times.

- (However, a thread having to give up its turn at the head of the queue and then going back to wait at the end of the queue after a false alarm can have **fairness** implications: see below.)
- If there are multiple threads waiting inside the same condition variable, you should make sure that whichever of them gets woken up is able to get out. For this reason, you must have a separate condition variable for each separate condition that your threads can be waiting for.
- If no thread is waiting for some condition inside that condition variable, **the unused `signal` does not get stored for the future awaits**, but vanishes as if it had never occurred. This can also cause tricky concurrency bugs if the threads do things in a different order than you had assumed.
- Each condition variable is always built on top of an existing mutex `Lock`. The thread must own that lock before it is allowed to call `await`. Therefore, **a thread can only `await` for a condition inside the critical section that uses that particular mutex lock**.
- When a thread enters the waiting state, **it gives up that mutex lock** (but not any others that it might currently own), so that other threads can enter that same critical section, which may be necessary to make that `condition` true. There is no risk of corrupted data due to race conditions, since the waiting thread is not doing anything, and thus is not reading and writing any data concurrently with any other threads.
- To actually wake up from the `await`, the thread must first recapture that mutex lock before it is allowed to proceed. If some other thread is currently the owner of that lock, the thread woken up has to wait in line with other threads for that lock to become available to it. This guarantees that at most one thread at a time can be active inside that critical section.

10.3. Fairness in concurrency

- As with many other terms in computer science and engineering, in the theory of concurrency the term "fair" has a rather different technical meaning than in our everyday language.
- In real life, we would not consider a restaurant waiting line or supermarket checkout queue where the doorman or cashier can at whim call some preferred customers to the front of the line past all the boring normos to be fair. But in computing, such a queue could be perfectly "fair".
- In the theory of concurrency, a system is said to be **fair** if no thread can ever **starve** by having to wait forever until it finally receives service. The actual waiting time is irrelevant in this binary theoretical sense: some thread having to wait a billion years ($O(1)$) for every request still makes the system "fair".
- **First-in-first-out (FIFO)** service by itself is obviously fair in this sense, assuming that the entity performing the service does not **deadlock**.
- In concurrency, a **deadlock** is created by **a cycle of waits** where each party is stuck waiting for the next one up the cycle to do something, typically resulting in too much mutex locking. Concurrent systems should be designed so that no such cycles can emerge, but this (like many other issues of concurrency) is an extremely nontrivial problem in general!
- Unlike the primitive object locks in the Java language that are not guaranteed to be fair, the objects of `Lock` (and other high level concurrency utilities in `java.util.concurrent`) can be

made fair (in fact FIFO, the Java concurrency API terminology is incorrect here) with a `boolean` constructor parameter.

- It is important to understand that **fairness does not propagate upwards**: building a concurrent composite using only fair parts does not guarantee the fairness of that composite system.
- Classic exercises in concurrency such as the **dining philosophers** or the **producers and consumers** illustrate the difficulties in designing concurrent systems that make deadlock and starvation impossible by design, and still be efficient and live.

10.4. Asking another thread to terminate

- Every thread will terminate by itself when it returns from its **top level call**, the `main` method for the main thread and the `run` method for others. But what if we decide that some thread should terminate earlier than that?
- For example, consider the graphics thread of some real time action game that runs in a loop that draws the game screen ~~25~~ 60 times a second. When the player quits the game, this thread should terminate along with all other threads of that program, so that the JVM process can terminate.
- Since calling the method `start` will start a thread once it has been created, it is probably not that difficult to guess which method will stop the thread. However, this method `stop` in `Thread` has long been officially **deprecated** and should never be used, since relying on this method is highly dangerous and error prone in any nontrivial program.
- Any resources held by the stopped thread are not released, its `finally` blocks are not executed, and anyway the thread is terminated in the middle of whatever operation it was currently at, possibly leaving the data in an inconsistent state.
- Instead of shooting the thread with the elephant gun of `stop`, we ask it nicely to stop at some convenient moment in the near future. The code executed by that thread is written to occasionally check the value of some `boolean` field (in all my code examples, this field is named `running`) shared by both the thread and the thread that might be asking it to stop.
- To stop a thread designed this way, just assign the value `running` to be `false` and leave the thread alone to eventually clean up after itself and terminate in proper order.
- Just one problem remains with this simple scheme: what if that thread happens to be stuck inside some condition variable, waiting for some condition that will never become true, since another thread that would have eventually made that condition true has already terminated as we were closing up the shop?
- Whenever you set `running` to `false`, you should also call the special method `interrupt` to all threads that are supposed to check whether they are still running. This method gives the thread a friendly "nudge" that is enough to **get it out from any wait** that the thread might currently be in. If the interrupted thread is not currently in any waiting state, the interrupt does nothing to the execution, except that it sets an internal `boolean` flag that will prevent it from entering a wait in the future.
- A good analogy to `interrupt` would be the way how an incoming casino dealer gives the previous dealer a light shoulder tap to let her know that the shift change is due and she should finish up the current deal, but not start another deal after that one.

- Instead of returning normally from that wait, the method throws an `InterruptedException`. This particular exception subtype happens to be checked, which is why all blocking methods in Java will declare this exception in their `throws` clause. The checked nature of this exception subtype is also the reason why the `run` methods must catch and handle it, since the `run` method overridden in the subclass of `Runnable` cannot let out any checked exceptions.
- A thread can also check (using the method `isInterrupted`) if it has been interrupted, and could theoretically be programmed to react to an interrupt whichever way you want. (After all, all syntactic things in programming really ever mean only whatever we say that they mean.) However, the most common and standard use for `interrupt` is to request termination.

10.5. Semaphores

- **Semaphores**, the great granddaddy of all concurrency control techniques, were historically the first high level concurrency control mechanism, invented by the late great **Edsger W. Dijkstra**.
- Semaphores are **universal** in the sense that all concurrency control could theoretically be done using those alone. Standard exercise in concurrency textbooks is to implement various other high level concurrency controls (e.g. `Condition`, `CountdownLatch`, `CyclicBarrier` or `BlockingQueue` in the Java terminology) using only semaphores, and vice versa.
- **A semaphore is an integer counter that can be initialized with any initial value**, positive or negative or zero. A semaphore is said to have a number of **permits** inside it. There does not exist any kind of `Permit` class or objects, we merely conceptually think of this integer as "permits".
- Two basic atomic operations called `acquire` and `release`, so that `acquire` will block until one permit becomes available (that is, the counter becomes positive) and then decreases the counter, whereas `release` unconditionally increases the counter without blocking.
- A semaphore with initial value of one can effectively work as a mutex lock, with the method `acquire` corresponding to `lock`, and the method `release` corresponding to `unlock`.
- A semaphore used as a mutex with initial value higher than one will permit that many threads to enter an **expensive section** simultaneously. This is a useful technique for **load control**.
- An important difference between semaphores and other controls such as locks and condition variables is that a **semaphore has no concept of owner**. At any time, **any thread can try to acquire or release a permit to the semaphore**. This flexibility is the source of the power of semaphores, but imposes some extra responsibility on the programmer.
- Surprisingly complex concurrency controls can be implemented using one or more shared semaphores with cleverly chosen initial values and logic to acquire and release them. (For those of you interested in this kind of stuff going potentially way beyond this course, check out the excellent "The Little Book of Semaphores" freely available online.)

10.6. Additional concurrency tidbits

- Before Java 5, locking of critical sections was done using the keyword `synchronized` that wrapped a block of code or an entire method to be synchronized with the particular object.
- In the Java virtual machine, **every object whatsoever can act as a synchronization lock**. This makes every object use more heap memory than it really ought to.

- Even more strangely, every object in the JVM heap is not only a synchronization lock, but an entire thread queue that can act as a condition variable. The methods `wait` and `notify` are inherited to every Java object from the universal superclass `Object` for this purpose.
- The class `Thread` also contains some more or less useful `static` methods for thread control. For example, to make some thread wait for some precise number of milliseconds, have it call the method `Thread.sleep(ms)` with the number of milliseconds to sleep given as parameter.
- Threads can be given a **priority**, a suggestion that the JVM should give the high-priority threads more execution time than it will give for the low-priority threads. However, these **priorities can only be used to optimize the speed of a concurrent system that is already correct for all its possible execution interleavings**, and they cannot make an incorrect program to be correct.
- Changing the thread priorities of a faulty concurrent system cannot help, since the JVM is not guaranteed to obey these priorities anyway. Even if it did, there is no guarantee that at some critical juncture, of the two waiting threads the JVM would always first wake up the higher-priority one, since that just might be the time for the lower-priority thread to get a turn to go first. (After all, even the lowest priority threads must *sometimes* get their turn to execute!)
- A single-threaded program is correct if it produces the correct answers for all possible inputs. **A concurrent program has to do that for every one of the infinitely many possible interleavings of its execution threads**. This makes concurrent programs and their behaviour vastly harder to reason about and debug than ordinary single-threaded programs.
- Concurrency bugs can be very annoying in that **they are usually not reproducible**, since they tend to manifest themselves only for a small portion of all possible execution interleavings, which are generally not reproducible even within the same machine.
- **Modifying a concurrent system to detect its bugs can change its timing and other behaviour** so that the bugs mysteriously vanish in that particular testing environment, and reappear when the detection is removed. Such a phenomenon is sometimes humoristically called a **Heisenbug**.
- Declaring a field to be `volatile` forbids the compiler from assuming that the field does not change unless explicitly assigned to. Such fields must always be read and written directly from memory at every use, and their values cannot be **cached** in the processor registers. This prevents various compiler optimizations such as **hoisting invariants** outside a loop, and whose correctness tacitly depends on the assumption that the fields cannot change unless explicitly assigned to.
- The package `java.util.concurrent.atomic` contains **atomic versions of boxed primitives** so that important non-atomic operations such as "increment and test" exist as internally atomic methods.

10.7. Concurrency and Swing

- Swing **runs in a separate event dispatch thread** that is not a daemon, and thus continues to exist even after the main thread and all other threads have terminated. Typically in a Swing program, the main thread merely runs the `main` method to set up the initial components before terminating, after which this Swing thread takes over.
- The event dispatch thread works as an infinite loop that processes the internal operations of AWT one at the time.

- **The event dispatch thread comes into existence when the first top level container becomes visible**, and terminates when there are no more visible top level containers. This makes the `setVisible` method call of `JFrame` way more important than it might initially seem.
- Note that the Swing event listener methods get executed in this AWT event dispatch thread. If your event listener does something that takes a very long time, this will freeze the entire Swing until that method terminates.
- If your event listener really needs to do something that takes a long time, it should create a new background task for that purpose, and then return immediately to allow Swing to continue its operation smoothly.
- If you create additional threads that concurrently modify Swing components, these modifications could interfere with the ordinary Swing operations, since Swing components are not guaranteed to be thread safe.
- To give Swing a `Runnable` task to execute in its event dispatch thread, use the utility class `SwingUtilities` and its methods `invokeAndWait` and `invokeLater`. Since Swing is guaranteed not to be doing anything else while it is executing tasks submitted that way, such tasks can safely modify all Swing components to their heart's content.

Lecture 11: Computation Streams and Lambdas

11.1. The increasingly important laws of Moore and Amdahl

- The famous **Moore's law** states that computing speed (or more accurately in the physical sense, **frequency**) doubles every 18 months. From its inception in 1965, this prediction has held up astonishingly well.
- Outside the field of astronomy and especially in any fields where humans and their interactions are involved, normally one would not observe numerical predictions that have unfolded in reality for over five decades as neatly as Moore's law, and never see such accurate predictions of the future anywhere else in engineering, let alone human and social sciences.
- However, this exponential growth cannot possibly continue forever, since there are hard both physical (speed of light, electronic noise, quantum uncertainty) and economic limits for it.
- Doubling the processor speed does not necessarily cut the running time of the program in half, unless reading and writing bytes to memory, often the bottleneck of machine code execution in real computers as opposed to our idealized Platonic virtual machine where each operation takes the same abstract O(1) time, is similarly accelerated.
- (Algorithms that access RAM bytes in the same memory page to minimize swapping and perform several calculations on the bytes they read from RAM into the processor registers before having to write the results out are said to have good data locality. For this reason, looping through a large 2D array using nested loops is faster with rows in the outer loop and columns in the inner, compared to having columns in the outer loop and rows in the inner.)
- A less painful way to speed up many computations is to **parallelize them over multiple processors and processor cores**. Unfortunately, computational problems vary greatly in how far they can theoretically be parallelized.

- **Amdahl's law** from the year 1969, not quite as famous and eerily prescient as Moore's law but still just as important even though people don't seem to realize it, points out that every computation consists of an **inherently sequential part** and **parallelizable part**. Using P idealized processors, the total running time will be $T = T_S + T_P / P$.
- At one end of this continuum lie the **inherently sequential problems** for which $T = T_S$. These problems consist of a strict series of **stages** so that the stage $n + 1$ cannot begin until the previous stage n has been completed.
- (Analogy: building a skyscraper, with the pouring of concrete to build each floor being a separate stage. Also the old chestnut of "Nine women can't team up to make one baby in one month.")
- The execution of the given arbitrary sequential program cannot in general be parallelized by algorithmic means, except in rare special cases. The lower level language some program is written in, the more difficult its parallelization becomes while guaranteeing the preservation of its higher level semantics.
- At the other end of this continuum reside the **embarrassingly parallel problems** for which we have $T = T_P / P$, at least in the idealized case. Such problems consist of a multitude of small tasks that are all independent of each other, allowing them all to be potentially evaluated in parallel. Typical examples would be all kinds of combinatorial **needle-in-haystack** searches and the **graphics rendering pipeline**.
- Graphics hardware speedup has obeyed its own accelerated version of Moore's law where the doubling happened every **six** months, which creates an eightfold increase over 18 months. Many other problems (such as **bitcoin mining** and **machine learning** with **neural networks**) have been found not to require the universal computation framework provided by the computer CPU, but can be pipelined and parallelized for GPU hardware that can then be piled in racks and warehouses to build a "poor man's supercomputer".
- Most actual problems and the algorithms to solve them fall somewhere between these two extremes in that they can be parallelized to some extent. An important part of the **science of algorithmics** these days is finding new ways to parallelize important computations.
- Parallelization works out ideally when the parallel processor cores do not need to communicate with each other, synchronize the access to modify any shared data, or wait for some previous tasks to finish and give us the result that we need to proceed. Synchronization of shared memory, especially if done in a crude "stop the whole world while I do my little thing" fashion, can slow down parallel computations significantly and even eliminate the time savings theoretically achieved by parallelization.
- But so it is in many other walks of life. For example, in software engineering, the famous **Brooks' Law**: "*Adding more manpower to a late project only makes it later.*"
- Another dimension to parallelize computations that consist of a large number of tasks, each task a strictly sequential series of stages (e.g. microprocessor machine code instructions, or computer graphics rendering pipeline) is to **pipeline** them so that each stage feeds results to the next one, and can start its next task immediately after passing the previous results to the next stage.
- This does not speed up the processing of one sequential task, but will greatly speed up the procession of, say, a million such tasks.

- Analogy: a factory assembly line, or a cafeteria line with separate stations for various foods, drinks and payment. Or the way Subway and similar fast food outfits pipeline and parallelize their assembly of sandwiches, with all sorts of small parallelization optimizations going on in there.

11.2. Fork-Join framework

- Java 5 improved the state of concurrent programming in Java by bringing in all sorts of high level concurrency controls such as `Lock`, `Condition` and `ExecutorService`. In this course, we will not even cover the language level concurrency techniques which are idiosyncratic to Java anyway, all concurrency controls should be as high level as the rest of the program.
- Ideally, the compiler and the execution environment should do the parallelization for us. Unfortunately, this requires language to be sufficiently high level to allow the necessary inferences about its semantics, otherwise parallelization is not guaranteed to maintain these semantics.
- Java 6 or 7 were not very big updates, but at least Java 7 introduced a higher level concurrency framework of a **Fork-Join framework** that hides away all the details of individual threads.
- A `ForkJoinPool` is a generalization of `ExecutorService` in which you can submit computations. These tasks have to be subtypes of `ForkJoinTask<T>`, or more usually its utility subclasses such as `RecursiveTask` where some methods are given a handy and meaningful default implementation.
- Just like with an `ExecutorService`, you only need one `ForkJoinPool` in your entire program.
- The `ForkJoinPool` keeps internally track of all thread maintenance and **load balancing of submitted tasks** to different threads, using as much computer resources to perform these tasks as the underlying machine provides. A Java program written today, if by some miracle executed by somebody in the far future, will fully use the power of future computers that will hopefully have hundreds or even thousands of parallel cores. ("Six hundred and forty cores should be enough for anybody.")
- Any `ForkJoinTask` can **fork** new tasks to existence to the pool, or **join** another task waiting for it to terminate. (In the theory of concurrency, every time you see the word "join", just mentally substitute "wait for termination", since that is what that word means. There must surely be some historical and technical reasons behind this term, but I don't know them.)
- It is your job to write your `ForkJoinTask` methods so that you try to happily `fork` as much as you possibly can, since the more you `fork`, the more parallelism you will get. In fact, every time you write a `fork`, you should mentally go "Yay! More parallelism for me!"
- Sometimes some particular task cannot proceed until some previous task has finished, in which case you have to write a `join`. Every time you `join`, you decrease parallelism, and should perhaps mentally go "Boo! Better recheck whether that join is really necessary."
- When solving an inherently sequential problem, you would never get to write a `fork`, so using a `ForkJoinPool` would be pointless. With an embarrassingly parallel problem, you never have to write a `join`, except possibly in the very end just before you return to say that everything is complete.

- A **semaphore** initialized to value `1-n` is an excellent technique to wait for `n` separate tasks to finish. The `acquire` operation for waiting will block until precisely `n` calls to `release` have taken place, regardless of the order in which these tasks complete their work.
- For problems that lie between these two endpoints of the continuum, you will write some forks followed by some joins. For example, the **parallel merge sort** can fork the subtasks of sorting the two halves of the array, but merging these sorted halves cannot begin until both halves have been sorted. (**Parallel quicksort** can similarly fork the subtasks of sorting the two partitions, with only one wait at the top level for every part to finish.)
- Since the pool needs to use both time and memory for task scheduling and maintenance, there necessarily exists some cutoff point so that **when a task is small enough, breaking it down to smaller tasks to be executed in parallel would cost more than the savings of parallelization**.
- Even when using a fork-join framework, with more complicated tasks sometimes you still need more fine-grained synchronization using locks, conditions and semaphores to protect the critical sections of your code, especially if your forked tasks access and modify shared objects.
- The Fork-Join framework also has no magical superpowers to divine the critical sections of code and data from your task submissions. Critical sections still depend on your intended higher level semantics, and are not inherent to the code itself.

11.3. Lambdas as syntactic sugar for anonymous classes

- **Statically compiled languages** tend to make a hard distinction between code and data, so that code is set in stone at compilation, whereas data can be created and modified dynamically. On the other hand, **dynamic and functional languages** use **homoiconic** representations for both so that **code is data and data is also code**, depending on which way you currently look at it.
- In the theory of programming languages, a **first class object** is (roughly) something that can be assigned to a variable or passed to and from a function or method. For example, strings, integers and birds are first class objects in Java, but **methods and blocks of code are not**.
- (The term “object” is used here in an older sense that predates the term “object-oriented programming”. Some material talks about “**first-class citizens of the language**” instead to dispel this confusion.)
- Strategy objects are often defined on the fly using an **anonymous class** that extends some **functional interface** that has only one method, the **functor object** thus wrapping that method inside a first class object that can be given to another method.
- **To pass a method to another method, pass a strategy object to the method.** This is ultimately just as good as passing the method for real as if we were working in some higher level functional language.
- Defined in the package `java.util.function`, there is a whole bunch of such functional interfaces such as `Function<T,R>`, `Predicate<T>`, `BinaryOperator<T>`, ...
- When you write the inner class that contains the method you want to pass as argument, you need to write a whole bunch of boilerplate to keep the compiler type checking happy. However, while tiring your fingers with such tedious typing chores more suited for our mechanized servants, you are not really telling the compiler anything that it doesn’t already know!

- Using the **lambda** operator `->` introduced in Java 8, you can essentially ask the compiler to fill in all that tedious boilerplate for you, so that you only need to type in the actual method body.
- For example, to create a strategy object that extends `Predicate<String>` and checks whether its parameter string is at least five characters long, just write `(x -> x.length() > 4)`. **The compiler will fill in everything else**, even the `return` keyword for such a one-liner!
- If the body has multiple statements, the curly braces are necessary. If the method takes more than one argument, their names must be listed in parentheses, e.g. `(x, y) -> x + y` for the anonymous function that adds its two parameters.
- What is the particular name of the method in `Predicate` or `BinaryOperator` that we were supposed to override in those examples? Who cares! The compiler knows it anyway, so have it look it up for us while we allocate our precious and limited brain cycles to more pressing matters.
- The term "lambda" refers to the historical syntax of defining an **anonymous function** that is itself data, with that greek letter used to denote the parameter variable. This dates all the way back to the thirties with **lambda calculus** of Alonzo Church.

11.4. Functional programming

- Ever since the early days of programming, languages split into two camps. One started from the machine and aimed high for mathematics, the other started from mathematics and aimed downwards for the machine.
- Most sources tend to consider these **imperative** and **functional** programming paradigms a black and white binary in a dualistic fashion, even though the reality is actually a continuum transitioning smoothly from one extreme to the other.
- The further towards the functional style you ascend in this continuum, the more you describe the problem to be solved, rather than describing the computational steps to solve that problem.
- In the imperative end of this continuum, each operation describes exactly what the physical computer actually does. The purest imperative programming is done in **assembly language** whose operations correspond directly to the operations of the processor. Most assemblers allow macros, eventually rising to the level of the **C programming language**, the lowest level high-level language still in common use.
- As languages ascend to talk about higher level abstractions of computation, they acquire more functional programming features. Different languages can therefore be roughly placed on this continuum based on how "high-level" they are.
- **You should not confuse "high-level" with high quality**: there are good and bad imperative languages (even at the low level realm, there are now Go and Rust to replace the old workhorse of C), just as there are good and bad functional languages. Every tool has its purpose, and **no tool can possibly be perfect for every purpose**.
- For any language and other programming tool, the proof of the pudding always ultimately lies in **whether real programmers can productively use it to solve their actual problems.**
- In fact, today we can get the best of both worlds by mixing languages of different levels to do what each one does best. For example, many games have their computation intensive and low level engine parts written in machine code or C, whereas the higher level logic is written with Python or similar high level language, making the game itself easier to script, extend and modify.

- Especially dynamic languages that can treat their own code as data and vice versa, instead of the code being set in stone at compilation, make for easy scripting for possibly entirely new set of rules for the entire game added dynamically! This is the approach and philosophy behind **data-driven programming**.
- Purely functional programming languages have been created as academic research projects, but in their strictness of immutable data and other functional programming aspects, they are too unwieldy and just flat out weird for real programmers to solve real problems. As the famous pithy expression goes, nobody has yet written an operating system kernel using a functional programming language.
- (Even writing silly arcade game of **Pacman** or **Space Invaders** that is supposed to run in real time, something that was achieved by a bunch of stoned long hair engineers back in the 1970's by soldering connections directly onto the motherboard, becomes non-trivial when using a pure functional programming language even with the best of our current tools. Something is therefore clearly very wrong with this entire picture.)
- (See also the post "Functional programming is not popular because it is weird" for a good and simple illustration of baking a cake to see why it is far easier for humans to think in forward imperative direction rather than backwards direction from definitions to execution.)
- We should also never forget that **no programming language can ever escape the physical reality of the underlying computer and its actual mindless low level operations**, no matter how high up in the outer space its concepts arise. In fact, losing sight of this underlying reality means losing sight of the true cost of your computations, akin to the Poor Shlemiel tale for writing nested loops when one would have sufficed, or in the field of software engineering, becoming an architecture astronaut whose designs essentially become highly convoluted ways of stating that "There exists some stuff, and then something happens."
- Old time functional programming languages were also unpopular due to their strange syntax (or in the case of Lisp, having essentially no syntax), but modern languages enable the functional programming ideas in a syntax that is more familiar to the imperative programming masses.

11.5. Meeting in the middle of the two paradigms

- It remains to be determined where exactly the "sweet spot" between purely imperative and purely functional programming lies. However, as the modern languages such as Python, Ruby, Rust, Kotlin etc. from this century demonstrate, this spot is clearly higher up the continuum than the spot where the major eighties and nineties languages such as C++ or Java reside.
- (Back in the day, even those familiar languages were considered "high level", except by a bunch of bearded Lisp and Smalltalk weirdos who just smirked and kept doing their own thing. See the section "The Blub Paradox" in the linked article from 2003 for a better description of the above notes, or the famous Greenspun's Tenth Rule of how programmers whose mindset is stuck to low level languages inadvertently end up crudely simulating higher level languages as soon as their project size grows past a certain threshold. See also "Frequently Rediscovered Technologies".)
- To cross the line from a static to a **dynamic** language, the language needs to have a built-in **eval**, the ability to **evaluate any dynamically generated string as an expression of that language**.

Having the eval furthermore allows the language to have a **REPL**, **read-eval-print-loop** for interactive use of that language. (Java 9 took the huge step of introducing both of these.)

- Functional programming **makes no distinction between code and data**, but either one can always be treated as the other. New code can be composed dynamically by combining existing functions in various ways. (Java 8 lambdas took a big step in this direction, even though the bytecode is still set in stone at compile time.)
- Furthermore, same as in mathematics, **functional programming treats all variables as final and all data as immutable**. This has multiple upsides, as we have seen many times already, but also the huge downside already encountered in the discussion of the autoboxing mechanism.
- Combined with pure functions with no **side effects** (other than possibly input and output to the outside world), immutability guarantees **referential transparency** in that the same expression will always evaluate to the same result, and any subexpression can always be replaced with a result-equivalent subexpression without affecting the value of the entire expression.
- Reading through an imperative program you can reason which computational steps can be taken, and in which order these steps happen. Functional programming languages are more free to reorder the evaluation, and more importantly for modern processors, **parallelize** it using the knowledge of the high-level structure to guarantee the correctness of this parallelization.
- Functional programming allows **partial evaluation** of some function to create another, more specialized function, which would be impossible in imperative programming languages with their strictly **eager evaluation**.

11.6. Computation streams

- Java 8 took a huge step towards the functional programming paradigm by introducing **computation streams** (not to be confused with I/O streams that we saw earlier, although conversions are trivial) using the **map-reduce framework**.
- Computations expressed in pure functional fashion with map-reduce streams **parallelize automatically** over multiple processors. Most famously, Google's search algorithms are internally written using this kind of framework, so that each individual query that would take well over a minute to execute sequentially in a single machine is broken down and parallelized to execute in a split second over thousands of servers in the massive server farm!
- A computation stream begins with some **supplier** of elements, and continues with various operations (most importantly `map` and `filter`) that perform computations on those elements. These operations can now be concisely written by using lambdas to define their strategy objects.
- A stream can be **sequential** or **parallel**, the latter being more efficient when the order of elements is irrelevant and the individual operations performed on them are **stateless**. A sequential stream guarantees to maintain the order of the elements in the stream and can therefore use **stateful** operations that depend on this order.
- An operation is **stateless** if its **result depends only on the current element coming from the stream**, but not the past history of previously processed elements. A stateless operation does not need to remember the past in its data fields, and can therefore be defined with a lambda, and computed in parallel for each element separately.

- For example, primality testing of integers is stateless, whereas maximum, average, sum etc. have to remember something of the past history to calculate the answers.
- A stream can be infinite, but is then cut down to a finite stream using a `limit` stage.
- In the other end of the stream, there is some kind of **consumer** that collects the elements and generates some kind of end result out of them, often using `reduce` to turn a stream of elements into a single element as the final answer.
- Many classes in the Java standard library (the entire Collection Framework, `File`, `Random` …) have been retrofitted to be able to supply a computation stream when asked. All iterators can also be converted back and forth to streams, to combine the power of both approaches.

11.7. Lazy evaluation

- **Lazy evaluation** is an important concept in **functional programming languages** where it can reside more comfortably and can even be baked into the very language semantics.
- Imperative programming languages are by their nature **eager** (for example, all arguments of a method call are always fully evaluated before the method call itself is executed), but some lazy evaluation ideas can also be applied when programming in the classic imperative style.
- Lazy evaluation means that **no computation is performed until it is certain that its value will definitely affect the end result**. This can potentially speed up computations by eliminating redundancy, although it can also make the computation have unpredictable pauses and use more memory in situations where the accumulated "debt" of delayed operations comes down for us to pay all at once, which makes such languages unsuited for **real-time applications** whose response time must be predictable.
- As a funny (but most likely apocryphal) tale of the definite pinnacle of lazy evaluation, the author recalls once reading about a professor who, instead of grading the final exams, simply failed everybody without even looking at the exams, and then actually graded the exam only for the small handful of students who came to his office to complain about their grades.
- I am not a lawyer nor do I play one on TV, but the difference between the civil law and common law legal systems feels analogous to the difference between lazy and eager evaluation.
- (In real word systems, different incentives and costs for different participants can distort the system in similar ways to produce arbitrarily perverse outcomes, as is illustrated by another, hopefully also apocryphal, anecdote of the commander of an aircraft carrier group who was trying to sleep in his cabin but was disturbed by the sunlight coming in through the porthole, so instead of just getting up to close the curtain, stayed in his bunk and intercommed in order for the entire carrier group to change its bearing.)
- Even in imperative languages, the **short circuiting boolean operators** and **if-else** are an example of lazy evaluation.
- In computer science, the word “lazy” does not have the same negative connotations that it has in the everyday language. In fact, an algorithm that is “lazy” in our everyday sense, that is, **does precisely that what is actually needed to produce the correct answer and not even one thing more**, would rather be called **elegant** or **optimal**! Somebody who could act like that in real life would surely be revered as a great zen master.

- As a real world analogy of lazy algorithmic thinking, imagine a raffle where the participants have to send in a postcard that must also contain the correct answer to the skill testing question. After the deadline, the winning card will be randomly picked from inside a spinning barrel.
- The eager evaluation version would check each postcard the moment that it arrives, and put the card in the barrel only if it had the correct answer. This guarantees that whichever card gets randomly picked will be legal, but also wastes time looking at all the cards that did not win and whose contents therefore cannot affect the end result.
- The lazy evaluation version will dump all postcards straight in the barrel without even looking at them. When the random winner is picked, only that card is checked for having the correct answer, and if it does not, just keep picking another random winner until the correct answer is found.
- Java **computation streams guarantee lazy evaluation** in that even though a stream is defined forward from supplier to consumer, its **evaluation is executed backwards so that each stage requests the previous state for an element only when it really needs an element**.
- Even if the producer or any intermediate stage produces a logically infinite stream, only a finite number of elements of that stream will ever actually get generated, precisely as many as the following stages actually need, no more and no less.
- If a field in an object is expensive to initialize but is needed only in rare situations, it can be **lazily initialized** to `null` (or some other convenient placeholder value, typically `0` or `-1`) and initialized the first time when its value is actually needed. For example, the expensive `hashCode` method in `String` is initialized lazily this way, and then **cached** for future use, knowing that **the hashcode of an immutable object cannot possibly change**.
- Lazy initialization can be vulnerable to **race conditions** in situations in which two threads try to use the same lazily initialized field simultaneously.

Bonus Lecture 12: Integers and Floats in Java

This entire topic is not a part of this course any more, but there is no cost in including these notes here for anybody who wishes to study them. Dealing with fixed-size integers and floating point numbers and understanding their limitations is an important and essential skill to avoid their silent pitfalls in all programming languages, not just in Java. Especially all the Pythonistas spoiled with having the luxury of unbounded integers in the core language and the `Fraction` type for exact representations of rational numbers in the standard library should take heed that not every language is a similarly blessed utopia where numbers work the way they are supposed to work...

12.1. Positional number systems

- Two fundamental principles: (1) An object is not the same thing as its name or representation, and (2) There is nothing magical about the number 10, except the universal agreement once upon a time made by the humanity of using base ten to represent integers basically just because we have ten **digits** in our hands.

- We represent integers in **a base 10 positional number system** so that each digit gives the magnitude of the power of 10 corresponding to that position. It is not enough just to see the individual digit, you must also see its position to know how much it is.
- Contrast this to **Roman numerals**, where X is always ten and M is always one thousand no matter where it is positioned. (A slight exception to this rule is that if a smaller unit comes before a larger one, its value is subtracted instead of added. For example, XI is 11, but IX is 9. However, canonically a smaller symbol can only be subtracted from the next larger one, so that 99 cannot be written out concisely as IC, but must be given in the for XCIX.)
- Every positive integer has exactly one unique representation in the positional number system, assuming that the leading zeros are ignored. (Proof by induction. No, newbie, this claim is not "trivially true". Hand-waved proofs such as "Every number has a unique representation in binary" are just begging the question.)
- By convention, the digits are written out left to right starting from **the most significant digit**. Many algorithms that operate on numbers would be easier to write as one-pass loops if the least significant digit came first, but convincing the entire humanity of the advantages of the reversed approach seems like a futile effort.
- Since there is nothing magical about number 10, we could use any other integer as base, and pretty much **everything would work exactly the same way**, including all numerical algorithms on sequences of digits that you already know (addition, multiplication, long division…)
- When using base n, the possible digits are always 0, …, n - 1.
- **Computers use internally binary base 2** for easier representation in electronics hardware.
- Binary numbers and the familiar base 10 integers are still the same integers as mathematical objects, just represented in a different way. The integer objects exist in the timeless platonic space of mathematical truths, and we humans use different ways to talk about these integers, the same way as different natural languages use different words to talk about the same things.
- **Mixed radix systems**, such as our familiar 24/60/60 representation for time, or the **imperial system of measures** for weights and lengths, use a different base for different positions. As long as the arithmetic is done properly with respect to the base of the current position, using a mixed radix system doesn't really change anything either.
- Bases 12 and 60 are actually more flexible than the base 10, since they can be easily divided into more useful parts than 10. Dividing things by either two or three is no problem in base 12.
- In principle, even negative numbers, complex numbers or Fibonacci numbers could be used as weird bases, although really only for **recreational mathematics** and similar amusement purposes for those of you with such inclinations. (See the works of the late great Martin Gardner.)

12.2. Bits and bytes

- The computer memory consists of a very large number of bytes, each byte comprising 8 bits. Everything that exists inside a **von Neumann architecture**, both code and data, must be somehow stored as bytes, since bytes are the only thing that physically exist in such an architecture, as far as we can reach down from our programming languages.
- An individual byte can store **an unsigned integer** in the small range from 0 to +255.

- **Bytes don't know what they represent**, or whether they have been grouped to represent larger structures. All semantics for the bytes is always imposed from the outside by whoever is currently using them. The exact same byte and its value in the computer memory can mean different things depending on whether that type is used as a machine code instruction, a part of a four-byte `int`, a part of an eight-byte `double`, or some larger data structure.
- We saw this same principle earlier with byte streams that assume or require nothing about the semantics of the raw bytes that they transport.
- To represent integers too big to fit into a single byte, we group together 2, 4 or 8 bytes to gain a wider range.
- The smaller the base, the longer the representation of a particular number. To make binary numbers easier to read for humans, **hexadecimal base 16** is used to pack a **nibble** of four bits into one **hexadecimal digit**. Letters A to F are conventionally used for the additional digits 10 to 15 for which do not have a numerical symbol for.
- Since the bases 2 and 10 do not have the same prime factors, changing one digit of the number inside one representation can change multiple digits (theoretically even all of them) in the representation in the other base. However, since 16 = 2^4, each hexadecimal digit corresponds exactly to one nibble and everything comes together perfectly.
- In Java and almost all other programming languages, hexadecimal integer literals are conventionally denoted with the prefix `0x`. Java 7 introduced the option of writing integers as **binary literals** with the prefix `0b`.

12.3. Signed integers and integer arithmetic

- To allow the representation of both positive and negative numbers, **signed integers** use the highest order bit to store the sign of that number, 1 being negative and 0 being nonnegative.
- Again, you cannot simply look at a byte or a group of bytes and determine whether it is signed or unsigned. Every byte means whichever way you decide that it means.
- To avoid having both positive zero and negative zero, which would make no sense for integers, the **two's complement signed representation** encodes -*n* by **negating the bits of the unsigned representation of *n*, and adding one**.
- Since zero is the odd man out in the middle, there is one more value to the negative direction than to the positive direction. For example, signed `byte` has the range -128, …, +127.
- **Java language standard requires** that `byte`, `short`, `int` and `long` absolutely positively must be stored as **two's complement signed integers** using 1, 2, 4 and 8 bytes, respectively. This guarantees that **integer arithmetic in Java will always produce exactly the same results in every platform that runs a standard compliant JVM installation**.
- Java does not have unsigned integer primitives in the core language. You know, "for simplicity", just like it still does not have **operator overloading**. (Because it is totes much simpler for everyone to read and write `a.add(b.mul(c))` than it is to read and write `a+b*c`.)
- **Integer arithmetic is always exact**, so that 2 + 2 is exactly 4, not one iota more or less. However, **integer arithmetic can overflow silently**, with no exception thrown and no way to detect later that an overflow took place.
- The class `Math` offers **checked integer arithmetic methods** that throw an exception at overflow.

- Utility class `BigInteger` represents signed integers whose size is limited only by the available heap memory, so each such number could have millions of digits. Arithmetic operations such as multiplication and division are performed with specialized algorithms (such as Karatsuba algorithm for integer multiplication) so that they are blazingly fast even for such humongous numbers.

12.4. Bitwise arithmetic

- Bitwise arithmetic operators are analogous to the logical operators for truth values, but performed separately in parallel for each bit of the two operands. Their importance is that they allow us to read and write individual bits of a number without affecting the rest of the bits in that number.
- Important use in **bit vectors** and related data structures that pack 32 bits into a single `int` value, using every part of every byte to store useful information. In Java, `EnumSet` and `boolean[]` are internally implemented as bit vectors.
- **Bitwise or**, denoted by `x | y`, produces a number where the *i*:th bit is on if it was on in **at least one** of the numbers `x` and `y`. (Just like for the logical or, this is an **inclusive or**.)
- **Bitwise and**, denoted by `x & y`, produces a number where the *i*:th bit is on if it was on in **both** of the numbers `x` and `y`.
- **Bitwise negation**, denoted by `~x`, produces a number where each bit is the **opposite from** what it was originally.
- **Bit shift operators** `<<` and `>>` move the bits left and right the given number of steps. For the right shift, `>>` maintains the highest order bit whereas `>>>` always brings in a zero.
- To turn on the *k*:th bit of `x`, use `x = x | (1 << k);`
- To turn off the *k*:th bit of `x`, use `x = x & ~(1 << k);`
- To read the value of the *k*:th bit of `x`, use `x & (1 << k) != 0`
- **Bitwise xor**, denoted by `x ^ y`, produces a number where the *i*:th bit is on if it was on in **exactly one** of the numbers `x` and `y`. Note that the caret is not exponentiation in Java.
- Since the **bitwise xor does not erase information** and **is its own inverse**, it has nice applications such as **one-time pad cryptography**, quite beautiful in their elegant simplicity.

12.5. Floating point

- In everyday life, we represent decimal numbers (in the sense of having a fractional part) in **fixed point decimal**, where the decimal point denotes the start of the negative powers of the base.
- For example, the two numbers 1.2345 and 1234.5 have the exact same **mantissa** of digits, but the latter is a thousand times larger than the former, as we can immediately see from the position of the fixed decimal point.
- Fixed point notation has two important weaknesses that make it unsuitable for representing decimal numbers in computations. First, it gives us **uniform precision** for the entire representable range, which is kinda silly, since the further away we move from zero, the less precision we need. Second, fixed point notation allows for a **relatively small maximum** value for the fixed length representation.

- In **scientific notation**, each number is rather expressed as the product of sign, mantissa and the power of the base. To make this representation unique for every representable value, the sign must be either +1 or -1, the mantissa has to be at least one but strictly less than the base 10, and the exponent must be an integer.
- The exponent tells us the **order of magnitude of that number**. The mantissa then places that number linearly within that order of magnitude, as if these numbers were lines of a uniform ruler.
- The previous example numbers 1.2345 and 1234.5 both have the same mantissa 1.2345, but the first number has the exponent of 0, whereas the second number has the exponent of 3.
- In Java and almost all other programming languages, scientific notation can be used to write constant literals in the form `1.234e-7`, the letter `e` standing for "times to the power of ten".
- Again, the number 10 has no magical properties that make anything work. **Floating point is simply scientific notation using base 2 instead of base 10**. **Nothing else changes.**

12.6. IEEE 754 floating point numbers

- **IEEE 754 standard** defines the **single precision** encoding where each floating point number is stored 32 bits by using 1 bit for the sign, 8 bits for exponent **biased** by +127 to guarantee that it is nonnegative, and 23 bits for the bits of the fractional part of the mantissa over the one.
- A handy trick to perform this conversion by hand for number *x* is to break it down into a sum of powers of two. Divide this sum by its highest term to produce the mantissa, and then multiply this highest term back to serve as the exponent. This way you can read the bits of the mantissa directly from the powers of two that appear in your breakdown of the mantissa.
- Instead of doing this by hand like some peasants, use sites such as IEEE 754 Floating Point Converter to play around with different numbers and their floating point representations.
- Similar **double precision** (hence, the type `double`) encoding for 64 bits uses 1 bit for sign, 12 bits for exponent and 51 bits for the mantissa. (There exists even 128-bit **quadruple precision** for special applications.)
- The standard also defines special bit patterns for `+0`, `-0`, `+Inf`, `-Inf` and `NaN`.
- When the exponent has its smallest possible value with the number being as close to zero as we can get, **subnormal** numbers have the mantissa values placed **logarithmically** along the real line interval, instead of the marks of a uniform ruler as is done for higher exponents. This makes arithmetic of extremely small positive numbers more useful and accurate.
- The `double` type can represent all positive and negative integers up to 2^{53}, and furthermore, all integers that are a product of some such number and some power of two up to 1023.
- The Java language standard requires that a `float` has to be stored **in at least 32 bits**, and a `double` has to be stored **in at least 64 bits**. However, this representation **is not required to be the IEEE 754 standard**, but other encodings can be used in special platforms. Floating point arithmetic in Java is therefore not guaranteed to produce identical results on different platforms, unlike integer arithmetic.
- Definitely the least used keyword in the Java language, `strictfp` forces the JVM to internally use the IEEE 754 standard for handling `float` and `double` values for a field, method or class. If this standard does not happen to be supported in some particular piece of iron, the JVM has to simulate the results on software, being as slow as if this were the 1980's again.

- (Terminology sidebar: **simulation** is done in software, whereas **emulation** is done on hardware.)
- `StrictMath` is otherwise the same as `Math`, but all its methods are defined to be `strictfp`.

12.7. Imprecision of floating point

- There exists an infinity of integers to both positive and negative directions, but we use only a small part of this infinity centered around the zero. Since the integers in that small part are “all there” so that nobody will ever discover a new integer between 7 and 8, binary representation allows us to represent all of them with arithmetic that is exact, unless there is an overflow.
- (As a lighthearted aside, can you explain why you are so sure that nobody will ever discover a new integer between 7 and 8 without resorting to any kind of circular reasoning, begging the question, proof by intimidation by energetic hand waving?)
- Decimal numbers are more difficult since there exists an **infinite continuum of numbers** between any two decimal numbers. So even if we restrict our attention to some small part of the infinite real line, there still exists the same infinite continuum of different numbers in there. The real number line is **holographic** in that the small part is the same as the entire thing.
- Fortunately, the floating point encoding is perfectly symmetric around the zero so that the number x is representable in floating point if and only if its negation $-x$ is representable. (This is not true for two's complement signed integers for which the negative side has one more value than the positive side, due to the zero being the odd man out in the middle.)
- No matter what the encoding, at most 2^{32} different numbers can be represented in 32 bits. There is no going past this fundamental limitation imposed on us by basic combinatorics. So, which numbers *should* we represent?
- A rather serious problem with floating point is that even many seemingly very simple numbers don’t have an exact representation in the base two floating point. (Even zero, the most important number of them all, had to be represented in a special way.)
- In base b, the fraction $1 / n$ has a terminating representation if and only if all the prime factors of n are also prime factors of b. For example, when b is the familiar base 10, we can represent 1/10 exactly as 0.1, but we cannot exactly represent 1/3, which would be 0.333…
- In base two, we can’t exactly represent **any** of the seemingly simple numbers 0.1, 0.2 or 0.3. All these would become infinite repeating series of negative powers of two.
- **Every program whose logic depends on equality or order comparisons of computed floating point values is broken by design**, and cannot possibly be fixed without a complete redesign of its entire logic.
- The only acceptable ordering comparison is comparison to zero. Any other comparison of floating point appearing in your code means you should use either `BigDecimal` or some kind of fraction type to perform your computations.
- **Do not use floating point to represent actual currency** or other decimal values that have to be exact in our everyday base ten. Always use `BigDecimal` to represent such values.
- A `BigDecimal` object consists of a `BigInteger` mantissa and an `int` exponent given in base 10, and all arithmetic is performed with this representation. This base can represent familiar numbers such as 0.1 exactly so that we can add ten cents to some dollar amount.

- Unlike in the primitive floating point numbers, this mantissa is not necessarily normalized, to account for differences in **precision** so that 0.3 and 0.30000 are different numbers that produce equal but different precision results in arithmetic operations.

12.8. Floating point arithmetic

- Even when the numbers `x` and `y` are exactly representable in floating point, the results of their simple arithmetic operations such as `x + y`, `x - y`, or `x * y` may or may not be exactly representable.
- Whenever floating point arithmetic would produce a result that cannot be exactly represented, the operations produce the closest representable number to the true result. If two results are equally close this way, the chosen result depends on the current **rounding mode**.
- If some long iterated computation is **chaotic**, even a small difference in some intermediate result can blow up to become a huge difference in the end result.
- Contrary to our intuition, floating point multiplication is easier and more accurate than floating point addition, especially when its operands are of vastly different orders of magnitude!
- Floating point hardware can internally use more bits of precision to guarantee the accuracy of the last bit of the mantissa of the multiplication result, even if the resulting mantissa is then truncated to 23 or 51 bits.
- Especially if `x` is several orders of magnitude larger than `y`, computing `x + y` becomes imprecise because for addition, `y` has to be **scaled up** to have the same exponent than `x`, which will necessarily cut off that many lower order bits from its mantissa.
- It can even be that `y` is too small to make any difference at all, so that `x + y == x`. In this case, adding `y` to `x` repeatedly even a trillion times inside some loop changes nothing!
- For this reason, floating point addition is **not commutative, although it is symmetric**.
- When adding lots of numbers, perhaps even billions of them, you should try to arrange the order of additions to keep the operands of each individual addition within the same general magnitude. (Alternatively, you can keep track of the current cumulative error with Kahan's algorithm.)