Primitives
Integral types: int, short, long, char, byte
Floating point types: double, float
Boolean types: boolean
- passed by value (copied when returned or passed as actual parameters)

Arithmetic Operators: ++, --, *, /, +, -
- Unary Operators: ++, --
- Binary Operators: *, /, +, -
- Operators of equal precedence evaluated left-to-right
- Difference in left-hand-value of prefix and postfix operators

Arithmetic Expressions
- Combination of arithmetic-valued operands and arithmetic operators
- Operands can be arithmetically-valued constants, variables, or method calls resulting in numeric types
- Resultant type of expression is the widest type of the operands of the expression

Assignment
- Assignment operator: =
- Left-hand side of assignment must be wider type than right-hand side (widening assignment)
- Narrowing assignments are illegal by default (to prevent potential data-loss), but can only be performed with a cast

Casting
- Allows a narrowing assignment by asking the Java compiler to "trust us"

Comparison Operators: ==, !=, <=, <, >, >=
- Result in a boolean value
- Only == and != can be used with object operands
- When used on two objects, == returns true if and only if two objects are the aliases of one another
  - .equals: Defined for semantic equivalence between two objects (programmer can define that even though two objects are not the same object, they should be treated equivalently)

Boolean Operators: !, ^, &&, ||
- Unary Operator: ! (Only one operand required)
- Binary Operators: ^, &&, || (Two operands required)
- Operands can be boolean-valued constants, variables, or method calls resulting in boolean types
- have precedence: (Arithmetic Operators) (Comparison Operators) ! ^ && ||
- Operators of equal precedence evaluated left-to-right

Boolean Expressions
- Combination of boolean-valued operands and boolean operators
- Result in a boolean value

Classes
Commonly model real-world entities inside applications
Templates for creating objects
Define what state objects will have (instance variables / fields)
Define behavior of objects (methods)
**Variables**
- Instance variables: defined within a class
- Local variables: defined within a method
- Variable scope: a variable is not accessible outside of the block in which it was declared

**Instance Variables** (a.k.a. *fields*)
- declared within class body, typically at the top
- have a type and a name
- type can be any primitive or class
- can restrict access with access modifiers
- are accessible in methods and constructors in the class

**Methods** (a.k.a. *functions*)
- can only be defined within classes
- can restrict use with access modifiers
- Accept input: parameters
- Produce output: return value

**Constructors**
- allow the programmer to define initialization of an object
- Java defines a default constructor for us, if we do not define one

**Overloading**
- Multiple versions of methods with the same name, with different method signatures

**Class Relationships**
- Dependency: Class references a method or field of another class.
- Aggregation: *Has-a* relationship between two classes. Class contains an instance variable whose type is another class. Aggregate object: an object that is composed of other objects
- Inheritance:

**Access Modifiers**: private, default (package protected), protected, public
- Define which entities can reference fields and invoke methods / constructors
- private: Only instances of this class
- default: Other classes in this package
- protected: Other classes in this package, or subclasses of this class
- public: Everyone, and their creepy uncle

**Objects**
- Instances of classes
- Have state that is independent of other instances of the class
- Passed by reference, *aliased* when returned or passed as actual parameters (changes enacted on objects inside a method are reflected on other aliases)
- The *this* reference refers to the currently executing object inside any method

**Object-Oriented Programming Concepts**
- Encapsulation and abstraction work hand-in-hand and are aided by access modifiers
- Encapsulation: prevent external entities from invalidating an object's internal state by defining how state can be accessed and preventing modifications to instance variables that would invalidate state. Encapsulation makes software more stable because it prevents program modules from interacting in ways that might cause incorrect / undefined behavior.
- Abstraction: present a simple interface for programmers to interface with, making your class or program module more easily usable (user doesn't have to understand internal representation). Abstraction makes software easier to write by providing complicated
components that can be easily used without fully understanding how they work.

**API**
- Application Programming Interface: The set public methods and fields that are provided to a programmer to use when programatically interacting with a class or program module.

**Predicates and Conditional Statements**
- if, else, switch, ?: (conditional ternary)
- Predicate: Based on boolean expressions
- Allow us to conditionally execute another statement (or block of statements)

**Loops / Repetitive Statements**
- repeat a statement (or block of statements) depending on the result of a boolean expression
- for loop: Best used when an exact number of iterations is known before hand
- enhanced for loop (a.k.a. foreach loop): For iterating over the elements of an array or an iterator
- while loop: Statements execute 0 or more times, best used when the number of iterations is unknown before hand
- do / while loop: Statements execute 1 or more times, best used when statements should be executed at least once

**Arrays**
- Arrays themselves are objects, and have a length property that tells us how many elements are contained in the array
- 0-based, valid indexes are 0 through array.length -1
- Indexing into an array with a non-valid index results in an ArrayIndexOutOfBoundsException
- Fixed in size
- Arrays of objects do not instantiate the elements of the array (all elements initially refer to null)

**DataStructures**
- Classes whose purpose is to store large amounts of data, i.e., other objects or primitives
- LinkedLists: Singly-Linked, doubly linked, most often implemented with Nodes, internal objects which carry content to be stored and references to previous and next elements
- Stacks: LIFO data structure
- Queues: FIFO data structure
- Maps: An association between a key and a value, where an easily recreated key is used to retrieve a value
- HashMaps (a.k.a. Hashtables): Implementation of map data structure with constant time performance, commonly implemented with an array of linked lists called buckets, elements are placed in buckets based on a hash function

**Iterators**
- Safe way to encapsulate and export a position within a data structure
- Commonly written as inner classes
- Allow multiple users to iterate or "step" over the data structure, looking at each element
- Most commonly provide next and hasNext methods. Other methods may vary with implementation: add, set, remove, etc.

**Recursion**
- See: Recursion (The worst CS joke ever)
- Base Case: The most simple cases of the problem, for which solutions are immediately
known
-Allows elegant solutions to a problem by breaking it into progressively simpler examples of
the problem, eventually ending in a base case
-The principle component of recursion in Java is the method
-Indirect recursion: recursive call spans multiple methods: m1->m2->m3->m1, etc.
-When to use instead of iteration: Is an iterative solution unclear? Can the problem be split
up into more simple cases?

Java Generics
-Allow the Java compiler to perform type-checking on
-Can eliminate the need for most type-casting
-Particularly useful in data structures
-With respect to type variables, throw what you know about inheritance out the window - a
LinkedList<String> is not assignable to a LinkedList<Object>, even though a String is
assignable to an Object. If this assignment were allowed, the LinkedList<Object> would
allow an Object to be inserted into a LinkedList<String>!
-Wildcards allow us to accept the widest range of compatible generic objects
-<T> extends T> We don’t know what type is actually stored in the generic class, but we
know that it is a T or some subtype of T. Useful when we want to read values of type T from
an object - we know at the very least that whatever is stored inside is a T, so we can store
values as Ts
-<T> super T> - We don’ know what type is actually stored inside of the generic class, but we
know that it is a T or some superclass of T. Useful when we want to be able to write objects
of type T to the generic class.

Exceptions / Errors
-Continually halt control flow of method upon occurrence and return control flow to calling
method, until handled or program terminates
-Checked: Typically caused by resources or input outside of the program's control: file
system, network, user input, require the programmer to acknowledge the possibility of an
exception being raised by 1.) defining handlers for the
-Unchecked: Usually caused by programmer error: NullPointerException,
ArrayIndexOutOfBoundsException, do not force the programmer to define handlers
-Allow the programmer to define code which attempts to deal with the exception in a try /
catch block
-Programmer can defer handling of an exception inside of a method by declaring that a
method throws the potential exception
-A programmer can take action regardless of the event of an exception or the presence of
handlers with thefinally clause, which is executed "no matter what"

Interfaces
-Allow for general treatment across multiple types of object with respect to non-core
concern
-Can only have public methods (methods are public by default if not specified)
-Can have constant variables (static final variables)
-We can have interfaces as variable types, but interfaces themselves cannot be instantiated
-Allows for multiple representation: implementing functionality internally in different ways,
all of which can be used with a common interface of methods (API)

Inheritance
-Represents an is-a relationship between classes
-Allows for code reuse and specialization (one class is a more specific version of another
class)
-Subclasses inherit all members (instance variables and methods) of the superclass
- All members are visible to the subclass, with the exception of private members
- Methods can be overridden to make behavior more specific
- Instance variables of a parent class can be *shadowed* by declaring another variable with the same name. This is generally discouraged.
- The `super` keyword can be used to invoke a class's immediate superclass's version of a method or constructor
- Constructors are not inherited
- Call to constructor must be the first line of a subclass's constructor
- Java will implicitly call default constructor of super class if not super constructor is called and a default constructor is visible

**Abstract Classes**
- Often represent abstract real-world concepts
- Cannot be instantiated
- Can be used as the type of a variable, but only more specific subclasses can be assigned to the variable
- Declared with keyword `abstract` in the class header
- Have 0 or more abstract methods

**Polymorphism**
- An object variable can potentially store many different types of object, so long as the type of the object is assignable to the type of the variable
- Objects of different types can be used generically based on a known, limited interface
- All object variables are potentially polymorphic
- Late Binding (a.k.a. *Dynamic Binding*) allows Java to invoke the implementation of a method as defined by the actual type of the stored object (as opposed to the declared type of the variable)
- Made possible by interfaces and inheritance

**Abstract Windowing Toolkit (AWT) vs. Swing**
- AWT was developed first, and relied on underlying operating systems to define the display of graphical components (Text fields, buttons, etc). This led to a non-standardized result on varying operating systems.
- Swing was developed to make GUIs more standardized across operating systems. It defines the low-level display of components for a more uniform cross-platform display, and as such, is slightly slower because Java is doing more work.
- Because many concerns of GUIs have no graphical representation, there was no need to reinvent these classes in Swing. Examples of classes with no graphical representation are LayoutManagers and classes for the event model (Listeners, Adapters, Events, etc).

**Java GUI**
- Components: (JButtons, JRadioButtons, etc.)
- Containers: (JPanels, etc.) *is-a* component
- Windows: *is-a* container, top-level, stand-alone window on desktop
- Frames: *is-a* window, top-level, stand alone window on desktop, has control bar and can have menus
- LayoutManagers: Define how components are positioned in a container
- Common LayoutManagers: FlowLayout (Default for JPanel), BorderLayout (Default for JFrame), GridLayout

**Event Handlers**
- Applications can choose to be notified of events (mouse / keyboard / button, etc) by subscribing classes
- Inner classes are particularly useful as event handling classes for 2 reasons: 1.) They have
access 2.) They are , making them perfect for limited functionality

**Inner Classes:**

**Nested Top-level Class** (Static member class)
- Top-level
- Referenced externally as: `ExternalClassName.NestedClassName`
- Declared inside of another class's class body with static modifier
- No access to members (fields and methods) of containing class

**Member Class**
- Declared inside of another class's class body
- Receive an *implicit* link to external, creating class
- Has access to all members of its containing class

**Local Class**
- Declared within a method or constructor
- Receive an *implicit* link to external, creating class
- Has access to all members of its containing class, and any final local variables in the method / constructor in which it is defined

**Anonymous Inner Class**
- Declared within a statement
- Does not have a name
- Receive an *implicit* link to external, creating class
- Must implement an interface or extend another class
- Inherits its constructor
- Does not define a constructor (because class is created at the same time as it is created)

**Threads**
- Create a new thread of execution (multiple simultaneous control flows in your application)
- Can be created by subclassing `java.lang.Thread` and implementing the `run` method
- Can be created by implementing `java.lang.Runnable` and passing your runnable object into a `java.lang.Thread` object
- Must be started by calling `start()`
- Terminate when the `run` method returns
- Simultaneous execution of threads is really a cruel trick played on us by the Java Virtual Machine (JVM), really what's happening is the JVM is giving each Thread a small amount of time in which to execute (a time-slice), alternating between Threads. This is one reason why Threads are not deterministic (we can't guarantee that things will execute in the order we think they should, especially for small time windows).
- Should be stopped by calling `interrupt()`. `interrupt()` does not actually force the thread to stop, but rather, informs it that it should stop as soon as possible. This allows for the thread to "clean up" after itself prior to terminating. The deprecated `stop()` method should not be used.

**Thread Safety**
- When multiple Threads access a shared resource (instance variable), a *race condition* can occur. In a race condition, multiple threads can read and write values to memory resulting in an unpredictable value for the shared resource, and thereby, unpredictable behavior.
- Can add the keyword `synchronized` to a method header to ensure mutually exclusive access by Threads
- The `synchronized` keyword can also be applied to a block statement
- Only one thread can be inside of a synchronized block on a particular object at once
- *synchronized* blocks can be overly restrictive and inflexible because they are bound to syntactical constraints of a block statement.
- Lock objects can allow programmers to have more control over when critical sections are protected (not bound to syntactical constraints). A Lock can conditionally be obtained or released, released in a downstream method, etc.
- Call *lock()* to obtain the Lock, *unlock()* to release it for other threads.
- Communication between threads using / waiting for a *Lock* object is made possible with *Condition* objects and the *await()* and *signal() / signalAll()* methods.

**Deadlock**
- When no thread can proceed because each is waiting on another to complete some task.
- Application (or some element of functionality) becomes unresponsive.