GenForge — Flashcards

A Spark & Anvil printable flashcard deck. Quiz yourself on the GenForge cast's curriculum — one card at a time.

How to use these cards

Fold each sheet down the middle line so the answers are hidden. Read the **question** on the left, say your answer out loud, then **unfold** to check the right side. Testing yourself — then checking — is one of the most powerful ways to remember. Get one wrong? That's the best kind of practice: try it again tomorrow.

Want real flashcards? Cut along the fold line and the rows into cards — question on one side, answer on the other.

---

1. Loops that generate visual and rhythmic patterns

✂ Question (fold →)	Answer
'Changing the loop count from 4 to 8 has no effect on the pattern.' This is:	**False — it doubles the number of repetitions (a denser/longer pattern)** — MISCONCEPTION NAMED — 'count doesn't matter.' The loop count directly controls how many times the body runs; changing 4 to 8 doubles the repetitions, producing a denser or longer pattern.
PREDICT: an inner loop draws 5 dots per row, and the outer loop repeats for 3 rows. Total dots:	**15** — 5 dots per row x 3 rows = 15 dots total — nested loops multiply their counts to fill a 2D grid.
PREDICT: 'repeat 3 times: {move forward, turn 120°}' draws (approximately):	**A triangle** — Repeating move-forward + turn-120° three times draws a triangle: three equal sides with three 120° turns (totaling 360°, a full rotation) closes the shape. Loop count = number of sides.
PREDICT: a loop rotates a shape by 30° and redraws it, repeated 12 times. The result is:	**12 copies arranged in a full circle (12 x 30° = 360°)** — Rotating by 30° and redrawing 12 times (12 x 30° = 360°) arranges 12 copies evenly around a full circle — a radial pattern, a classic generative-art result of looped rotation.
NESTED loops (a loop inside a loop) are useful for:	**Creating 2D patterns like grids** — Nested loops (one loop inside another) generate 2D patterns like grids — the outer loop handles rows, the inner loop handles columns, drawing a shape at each combination.
To draw a regular polygon with N sides in a loop, each turn should be:	**360° / N** — For a regular N-sided polygon, the total turning is 360°, split evenly among N corners, so each turn is 360°/N (e.g., 90° for a square, 72° for a pentagon).
PREDICT: an outer loop of 3 containing an inner loop of 4 runs the innermost body how many times total?	**12 times (3 x 4)** — With nested loops, the inner body runs (outer count x inner count) = 3 x 4 = 12 times total — e.g., drawing a 3-row by 4-column grid of 12 shapes.
PREDICT: a loop plays a drum hit every beat for 8 beats. The result is:	**8 evenly-spaced drum hits** — A loop playing a drum hit each beat for 8 beats produces 8 evenly-spaced hits — the loop generates the steady rhythmic pattern by repeating the hit once per beat.
In music, a loop can create:	**A repeating rhythmic or melodic pattern** — In music coding, a loop repeats a rhythmic or melodic pattern — creating grooves, ostinatos, and beats by playing a sequence of notes/hits over and over.
PREDICT: a loop that increases a circle's size by 10 each iteration, run 4 times starting at size 10, ends with a circle of size:	**50 (10 + 4×10)** — Starting at 10 and adding 10 on each of 4 iterations gives 10 + 40 = 50 (the sizes drawn are 20, 30, 40, 50). Loops can accumulate a changing value across iterations.
'A loop always runs forever.' This is:	**False — a loop runs for its defined number of iterations (or until its condition ends)** — MISCONCEPTION NAMED — 'loops run forever.' A loop runs for its defined iterations or until its condition becomes false. An unintended infinite loop is a BUG; a well-formed loop terminates.
A LOOP in code is used to:	**Repeat a set of instructions multiple times** — A loop repeats a set of instructions multiple times — the core tool for generating patterns (e.g., drawing

	many shapes) without writing each repetition by hand.
PREDICT: 'for i from 0 to 4: draw a dot at x = i x 20' draws dots at x-positions:	**0, 20, 40, 60, 80** — With i = 0,1,2,3,4 and x = i x 20, the dots land at 0, 20, 40, 60, 80 — evenly spaced. Using the loop variable in a calculation spreads elements across space.
A loop variable (like 'i') can be used inside the loop to:	**Vary each iteration (e.g., change position or size)** — The loop variable (e.g., 'i' going 0,1,2...) can be used inside the loop to vary each iteration — offsetting position, scaling size, or shifting color — creating evolving patterns rather than identical repeats.
PREDICT: 'repeat 6 times: {move forward, turn 60°}' draws:	**A hexagon (6 sides)** — Six iterations of move + turn-60° (6 x 60° = 360°) draws a hexagon — six equal sides. The loop count equals the number of sides, and the turn angle is 360°/6 = 60°.
A loop that runs 0 times will:	**Not execute its body at all** — A loop set to run 0 times skips its body entirely, producing no output from it. The iteration count can legitimately be zero, in which case nothing inside the loop happens.
Iteration is powerful in generative art because it:	**Produces complex patterns from simple repeated rules** — Iteration lets simple repeated rules generate rich, complex patterns automatically — a hallmark of generative art, where a short loop can produce intricate visual or rhythmic structures.
PREDICT: a loop 'for i from 1 to 4' runs its body how many times?	**4 times** — A loop 'for i from 1 to 4' runs with i = 1, 2, 3, 4 — four iterations. Counting the values in the inclusive range gives the number of repetitions.
Understanding iteration and loops lets a creative coder:	**Generate complex visual and musical patterns from concise repeated code** — Understanding iteration/loops lets a creative coder generate complex visual and musical patterns from concise, repeated code — predicting the output of a loop is the foundation of generative art and music.
PREDICT: 'repeat 5 times: draw a circle' produces:	**5 circles** — The loop body ('draw a circle') runs once per iteration; repeating 5 times draws 5 circles. The number of iterations determines how many times the body executes.

2. Reusable procedures that produce many variations

✂ Question (fold →)	Answer
PREDICT: a function 'tree(depth)' draws a fractal tree; larger depth = more branches. Calling tree(2) vs tree(5) gives:	**tree(5) is more complex/branchy than tree(2)** — With 'depth' controlling recursion/branching, tree(5) produces a more complex, branchier fractal than tree(2). The parameter tunes the output's complexity — one function, a family of trees.
Understanding functions and parameters lets a creative coder:	**Build reusable, flexible procedures that generate endless variations** — Understanding functions and parameters lets a creative coder build reusable, flexible procedures — one form yielding many variations through different arguments — the key to efficient, expressive generative art and music.
A function can call ANOTHER function to:	**Build complex results from simpler building blocks** — A function can call other functions, composing simple building blocks into complex results — e.g., drawFlower() calls petal() several times. This composition is central to organizing creative code.
Using well-named functions makes creative code:	**Easier to read, reuse, and modify** — Well-named functions (drawFlower(), fadeColor()) make code easier to read (the name states intent), reuse, and modify — organizing a program into understandable, reusable pieces.
An ARGUMENT is:	**The actual value passed to a parameter when calling a function** — An argument is the actual value supplied for a parameter when a function is called — in drawStar(50), 'size' is the parameter and 50 is the argument. Different arguments yield different results.
A RETURN VALUE from a function is:	**A result the function sends back to be used** — A return value is a result a function computes and sends back to the caller (e.g., a function that returns a mixed color). It lets functions produce values, not just perform actions.
'A function can only be called one time.' This is:	**False — a function can be called as many times as needed** — MISCONCEPTION NAMED — 'a function runs once.' The whole point of a function is reuse — it can be called any number of times, often with different arguments to produce variations.
Functions and loops together let a coder:	**Generate many parameterized variations efficiently** — Combining functions (reusable, parameterized behavior) with loops (repetition) lets a coder efficiently generate many variations — e.g., looping calls to a shape function with changing arguments to build a whole composition.
PREDICT: 'drawSquare(size)' is called with size = 50, then size = 100. The result is:	**Two squares — one small, one large** — Calling drawSquare(50) then drawSquare(100) produces two different-sized squares — the parameter 'size' changes the output each call. One function, many variations.
PREDICT: 'note(pitch, duration)' plays a note. Calling note(C, 1) then note(E, 1) then note(G, 1) plays:	**Three notes: C, E, G in sequence** — Calling note(C,1), note(E,1), note(G,1) plays the three notes C, E, G in sequence (each for duration 1) — a parameterized function generating a melody through successive calls.
Which best describes a FUNCTION WITH A PARAMETER?	**'drawHexagon(size)' — one procedure that draws a hexagon of any given size** — 'drawHexagon(size)' is a function with a parameter — a reusable named procedure that draws a hexagon of whatever size is passed in, producing many variations from one definition.

A FUNCTION in code is:	**A reusable named block of instructions** — A function is a reusable, named block of instructions. Defining it once lets you run (call) it many times by name — organizing code and avoiding repetition.
'Parameters make a function less flexible.' This is:	**False — parameters make a function MORE flexible (one function, many outputs)** — MISCONCEPTION NAMED — 'parameters reduce flexibility.' Parameters INCREASE flexibility: a single parameterized function can produce endless variations (different sizes, colors, positions) instead of being locked to one fixed output.
PREDICT: 'drawCircle(x, y, r)' called as drawCircle(100, 50, 20) draws a circle:	**at position (100, 50) with radius 20** — Arguments map to parameters by position: x=100, y=50, r=20 — a circle centered at (100, 50) with radius 20. Argument ORDER matters and must match the parameter order.
A PARAMETER of a function is:	**An input that lets the function behave differently each call** — A parameter is an input to a function that lets it produce different results on each call — e.g., drawStar(size) with 'size' as a parameter draws stars of varying sizes from one function.
PREDICT: 'star(points)' draws a star with the given number of points. star(5) and star(8) produce:	**A 5-pointed star and an 8-pointed star** — star(5) draws a 5-pointed star and star(8) an 8-pointed star — the parameter 'points' generates different variations from one function definition.
PREDICT: a function 'petal(angle)' draws one petal at the given angle. Calling it in a loop for angles 0, 72, 144, 216, 288 produces:	**A 5-petal flower** — Calling petal(angle) at 0, 72, 144, 216, 288 (evenly spaced by 72° = 360°/5) draws 5 petals radiating out — a 5-petal flower. Combining a parameterized function with a loop generates rich patterns.
A function with MULTIPLE parameters like 'rect(width, height)' lets you:	**Control several aspects of the output independently** — A function with multiple parameters (e.g., rect(width, height)) lets you control several aspects independently — varying width and height separately to draw many different rectangles from one function.
Changing a function's DEFINITION affects:	**All calls to that function** — Editing a function's definition changes the behavior of ALL its calls — a key advantage: fix or improve once, and every place that uses the function benefits.
The benefit of using a function (vs copying code) is:	**Reuse and easier changes (edit once, affects all calls)** — Functions enable reuse: you define behavior once and call it many times. Changing the function updates every call — far easier to maintain than copying and editing repeated code.

3. Storing and changing values over time

✂ Question (fold →)	Answer
PREDICT: a variable 'y' starts at 0 and increases by 2 each frame. After 4 frames, y is:	**8** — Starting at 0 and adding 2 per frame for 4 frames gives 8 (values 2, 4, 6, 8). A variable that changes each frame drives animation — this is evolving state.
'Once you set a variable, its value can never change.' This is:	**False — variables can be reassigned/updated** — MISCONCEPTION NAMED — 'variables can't change.' Variables VARY — that's their purpose. You can reassign them anytime (a CONSTANT is what stays fixed). Updating variables is how programs track changing state.
'Two variables with the same value are the same variable.' This is:	**False — they are separate containers that happen to hold equal values** — MISCONCEPTION NAMED — 'equal values = same variable.' Two variables are separate containers; they can hold equal values but are independent. Changing one doesn't affect the other (for simple values).
PREDICT: 'count = 0' then a loop runs 'count = count + 1' five times. Final count:	**5** — Starting count at 0 and adding 1 five times yields 5 — a counter variable accumulating across loop iterations, a very common pattern for tracking how many times something happened.
A variable that increases by a fixed amount each frame creates:	**Steady (linear) motion or growth** — A variable increasing by a fixed amount each frame produces steady linear motion or growth (e.g., x += 5 moves an object at constant speed) — the basis of simple animation.
A BOOLEAN variable stores:	**A true/false value** — A Boolean variable stores true or false — useful for flags/switches (e.g., isPlaying = true) that control whether something happens, tracking a two-state part of the program's state.
A variable used as a POSITION (like x, y) that updates each frame produces:	**Movement/animation** — Updating a position variable (x, y) each frame and redrawing produces movement/animation — the object appears to move as its stored position changes over successive frames.
In music code, a variable could store:	**The current tempo, volume, or note being played** — In music coding, a variable can store musical state — the current tempo, volume, key, or note — and changing it alters the sound over time (e.g., increasing tempo variable to speed up).
PREDICT: 'size = 20; size = size x 2' — the final size is:	**40** — size = size x 2 takes the current value (20) and doubles it, storing 40. Variables can be updated with any arithmetic, not just addition.
PREDICT: a variable 'brightness' starts at 100 and decreases by 25 each step. After 3 steps, brightness is:	**25** — Starting at 100 and subtracting 25 for 3 steps gives 25 (values 75, 50, 25). A decreasing variable can drive a fade-out effect — evolving state over time.
PREDICT: 'total = 0' then a loop adds each of the values 5, 10, 15 to total. Final total:	**30** — An accumulator variable starting at 0 and adding 5, then 10, then 15 ends at 30 — accumulating a running sum across iterations, a fundamental use of a state variable.
PREDICT: 'hue = 0' and	**A color that cycles smoothly through the spectrum** — Increasing a hue variable by

each frame 'hue = hue + 10', wrapping at 360. This creates:	10 each frame (wrapping at 360) cycles the color smoothly through the spectrum — an evolving state variable driving an animated color effect.
PREDICT: 'a = 3; b = a; a = 10' — the value of b is:	**3 (b copied a's value BEFORE a changed)** — When b = a runs, b copies a's current value (3). Reassigning a to 10 afterward does NOT change b — b is still 3. For simple values, assignment copies the value at that moment.
Tracking state with variables is essential for:	**Animation, interaction, and any behavior that changes over time** — Variables tracking state are essential for animation (positions change), interaction (respond to input), and any evolving behavior — the program needs to remember and update values over time.
A CONSTANT differs from a variable in that a constant:	**Is meant to stay fixed (unchanging)** — A constant is a named value meant to stay fixed throughout the program (e.g., PI, a canvas size). It contrasts with a variable, whose value is expected to change.
Variables let a program:	**Remember and change values as it runs** — Variables let a program remember values and change them as it runs — enabling responsiveness, animation, accumulation, and any behavior that depends on evolving state.
STATE refers to:	**The current values of a program's variables at a given moment** — State is the collection of current variable values at a moment in time — the program's 'situation.' In animation, state (positions, colors) changes each frame to create motion.
Understanding variables and state lets a creative coder:	**Store, update, and use changing values to drive animation and behavior** — Understanding variables and state lets a creative coder store, update, and use changing values — driving animation, interaction, and any behavior that evolves over time. Predicting how state changes is core to reading dynamic code.
A VARIABLE in code is:	**A named container that stores a value** — A variable is a named container storing a value (a number, color, position). You can read it, use it, and update it — variables hold the changing STATE of a program.
PREDICT: 'x = 10; x = x + 5' — the final value of x is:	**15** — After x = 10, the statement x = x + 5 takes the current value (10), adds 5, and stores 15 back in x. Reassignment updates a variable's value.

4. Using randomness deliberately (and reproducibly)

✂ Question (fold →)	Answer
PREDICT: sizes are 'random(10, 30)' for 100 circles. The circles will be:	**Various sizes between 10 and 30** — Assigning each circle a size of random(10, 30) gives 100 circles of various sizes between 10 and 30 — randomness within a range produces controlled variation.
PREDICT: the same seeded program is run, then the seed is CHANGED and it's run again. The two outputs are:	**Different from each other (each seed = its own result)** — Changing the seed produces a different result (each seed generates its own reproducible sequence). So exploring different seeds is how a generative artist searches for outputs they like — each seed a distinct, repeatable variation.
In creative coding, RANDOMNESS is used to:	**Add variation and unpredictability to generated art/music** — Randomness introduces variation and unpredictability into generative art/music — scattering elements, varying colors, or improvising rhythms so outputs feel organic and non-repetitive.
PREDICT: 'random(0, 1)' used to decide 'if > 0.5, draw a circle, else a square' will:	**Draw circles and squares in an unpredictable mix** — Using random(0,1) with a 0.5 threshold makes a coin-flip-like choice each time, producing an unpredictable mix of roughly half circles and half squares — randomness driving branching decisions.
'random(1, 10)' typically returns:	**A random number between 1 and 10** — random(1, 10) returns a random number within the range 1 to 10 — a different value on each call (unless seeded), letting a program pick varied values within defined bounds.
Which best describes CONTROLLED (constrained) randomness?	**Random choices made within deliberate limits (range, palette, scale)** — Controlled randomness makes random choices within deliberate limits — a value range, a color palette, a musical scale — producing varied yet coherent output. It's neither total chaos nor fully fixed.
In music, controlled randomness can:	**Generate varied but musically-sensible melodies (e.g., random notes from a scale)** — In music, constraining randomness — e.g., picking random notes from a scale rather than any pitch — generates varied but musically-sensible melodies, balancing surprise with coherence.
A 'random walk' (each step moves a random small amount) produces:	**A wandering, organic path** — A random walk — where each step adds a small random offset to the position — produces a wandering, organic, meandering path, a classic generative technique for natural-looking lines and textures.
'Every run of a program using random() gives the same result.' Without a fixed seed, this is:	**False — unseeded randomness usually differs each run** — WITHOUT a fixed seed, random() typically produces different results each run (often seeded from the clock). Only a FIXED seed makes runs identical. Unseeded = varied per run.
To reproduce a generative artwork exactly, you must record:	**The seed (and the code)** — To reproduce a seeded generative artwork exactly, record the seed and the code — together they deterministically regenerate the same output. The seed is the reproducible 'recipe.'
Constraining randomness (e.g., random within a range or palette) is useful to:	**Keep the output varied but still coherent/controlled** — Constraining randomness — picking random values within a chosen range or from a set palette — keeps output varied yet coherent and controlled, avoiding chaotic or off-palette results while retaining organic variety.

PREDICT: two runs use the SAME seed and the same code. The random results will be:	**Identical between the two runs** — With the same seed and code, the 'random' sequence is identical each run — so the generated art/music is reproducible. This is the key property of SEEDED (controlled) randomness.
Understanding generative and controlled randomness lets a coder:	**Create varied, organic, and reproducible generative art and music** — Understanding randomness — using it within constraints and controlling it with seeds — lets a creative coder produce varied, organic, AND reproducible generative art and music, predicting how seeded randomness behaves.
Pseudo-random numbers in code are:	**Generated by a deterministic algorithm that appears random** — Computer 'random' numbers are pseudo-random — produced by a deterministic algorithm from a seed. They appear random but are fully reproducible given the same seed (which is why seeding works).
Why would a generative artist use a SEED?	**To reproduce a favorite random output exactly (and share it)** — A generative artist uses a seed to reproduce a specific 'random' output they liked — since the same seed regenerates it exactly. This makes randomized art reproducible and shareable (share the seed, share the piece).
PREDICT: colors are chosen 'random from [red, blue, green]'. The result uses:	**Only red, blue, and/or green (randomly)** — Choosing 'random from [red, blue, green]' restricts choices to those three colors — randomness within a defined set. The output is varied but stays within the chosen palette.
PREDICT: 'draw 50 dots at random positions' produces:	**A scattered, unpredictable arrangement of 50 dots** — Placing 50 dots at random positions produces a scattered, irregular arrangement — randomness spreads elements unpredictably, giving an organic, non-uniform look (unlike a deterministic grid).
Combining randomness with a loop lets a coder:	**Generate many varied elements automatically** — Combining a loop with randomness generates many varied elements automatically — e.g., a loop drawing shapes at random positions, sizes, and colors creates a rich, organic generative composition.
A SEED in a random generator is:	**A starting value that makes the random sequence REPRODUCIBLE** — A seed is a starting value for a random-number generator; using the SAME seed reproduces the SAME sequence of 'random' numbers — making generative output deterministic and repeatable.
'Seeded randomness is truly unpredictable and never repeats.' This is:	**False — a fixed seed makes the sequence reproducible (deterministic)** — MISCONCEPTION NAMED — 'a seed makes it more random.' A fixed seed makes randomness REPRODUCIBLE — the same seed always yields the same sequence. Seeding is how creative coders get deterministic, shareable 'random' art.

5. Representing and generating color programmatically

✂ Question (fold →)	Answer
In HSB/HSL color, the HUE component controls:	**The base color (position on the color wheel)** — In HSB/HSL, hue controls the base color (position on the color wheel, often 0-360°); saturation controls vividness and brightness/lightness controls how light or dark it is. HSB is often more intuitive for generating colors.
Choosing random colors FROM A PALETTE (vs fully random) tends to produce:	**A cohesive, harmonious look** — Picking random colors from a curated palette produces variation that still looks cohesive and harmonious — unlike fully random colors, which can clash. Constraining color to a palette is key to controlled generative aesthetics.
PREDICT: RGB(0, 0, 0) is _ **and RGB(255, 255, 255) is** _:	**black; white** — In additive RGB, all channels at 0 (no light) is black; all at 255 (full light) is white. The extremes of the RGB model.
PREDICT: mapping a shape's SIZE to its color (bigger = brighter) creates:	**A visual link between size and brightness across the piece** — Mapping a property (size) to color (brightness) creates a visual link — larger shapes appear brighter — so color encodes information. Data-driven color mapping is a powerful generative/visualization technique.
Understanding color in code lets a creative coder:	**Represent, generate, and harmonize colors programmatically** — Understanding color models (RGB/HSB), palettes, alpha, and gradients lets a creative coder represent, generate, and harmonize colors programmatically — a core expressive dimension of generative art.
PREDICT: drawing many overlapping shapes with LOW alpha (high transparency) creates:	**Soft, blended color buildup where they overlap** — Drawing many low-alpha (transparent) shapes builds up soft, blended color where they overlap — the transparency accumulates, creating gradients and depth. A common generative technique for organic, layered textures.
To generate COMPLEMENTARY colors in code, you would:	**Pick hues ~180° apart on the wheel** — To generate complementary colors, choose hues about 180° apart on the wheel (e.g., 30° and 210°) — using the hue axis to compute opposite, high-contrast color pairs.
PREDICT: RGB(255, 255, 0) (full red + full green, no blue) is:	**Yellow** — In additive (light) color, red + green = yellow, so RGB(255, 255, 0) is yellow. Additive mixing on screens differs from mixing pigments (where red+green would muddy).
'RGB and HSB describe completely different colors.' This is:	**False — both describe the same colors using different coordinate systems** — MISCONCEPTION NAMED — 'RGB and HSB are different colors.' They're two coordinate systems describing the SAME set of colors; any color has both an RGB and an HSB representation. HSB is just often more intuitive for generating/varying colors.
To generate a set of ANALOGOUS colors in code, you would:	**Pick hues close together on the color wheel** — To generate analogous colors, pick hues close together (small hue steps, e.g., 200°, 210°, 220°) — using the HSB hue axis to select neighboring, harmonious colors programmatically.
A GRADIENT in code is:	**A smooth transition between two or more colors** — A gradient is a smooth transition between colors, generated by interpolating (blending) between them across space — e.g., fading from blue at the top to orange at the bottom of a canvas.

PREDICT: interpolating from RGB(0,0,0) to RGB(255,255,255) across a row produces:	**A black-to-white grayscale gradient** — Interpolating from black (0,0,0) to white (255,255,255) across a row produces a smooth grayscale gradient — each channel rising together from 0 to 255. Interpolation blends between two colors.
Which approach best keeps a generative piece's colors COHESIVE?	**Constrain colors to a defined palette or a narrow hue range** — Constraining colors to a defined palette or a narrow hue range keeps a generative piece cohesive. Fully random color across the whole spectrum tends to clash and look chaotic.
PREDICT: a loop increases HUE by 30° each iteration for 12 iterations. It generates:	**12 colors spanning the full color wheel** — Increasing hue by 30° for 12 iterations (12 x 30° = 360°) generates 12 colors evenly spaced around the full color wheel — a rainbow. Looping hue is a common way to generate a spectrum palette.
A PALETTE in generative art is:	**A defined set of colors the artwork draws from** — A palette is a defined set of colors an artwork uses. Constraining generation to a palette keeps the output cohesive and intentional, even when other aspects are randomized.
In code, an RGB color is defined by:	**Amounts of Red, Green, and Blue** — In code, an RGB color is defined by the amounts of Red, Green, and Blue light (each typically 0-255). Combining these channels produces the full range of screen colors (additive color).
Using HSB makes generating a rainbow easy because you can:	**Loop the hue while keeping saturation and brightness fixed** — HSB makes rainbows easy: loop the hue from 0 to 360 while keeping saturation and brightness fixed, and you sweep through the full spectrum at consistent vividness — much simpler than juggling RGB channels.
'Changing only the hue value keeps a color equally bright and vivid.' In HSB this is:	**True — hue is independent of saturation and brightness** — In the HSB model, hue, saturation, and brightness are independent axes — changing only hue rotates the color around the wheel while keeping saturation and brightness the same. This makes cycling colors easy.
An ALPHA value in a color controls its:	**Transparency/opacity** — The alpha value controls a color's transparency/opacity (0 = fully transparent, max = fully opaque). Lower alpha lets colors layer and blend, creating depth and soft overlaps in generative art.
PREDICT: RGB(255, 0, 0) is:	**Pure red** — RGB(255, 0, 0) is maximum red with no green or blue — pure red. Reading RGB values means interpreting the three channel amounts.

6. Translating, rotating, and scaling shapes

✂ Question (fold →)	Answer
Which sequence draws a shape moved right AND rotated?	**Translate right, then rotate, then draw** — Translating right and then rotating before drawing moves and orients the shape (though the order affects the exact result). Applying both transforms before the draw call combines their effects.
PREDICT: translating by (50, 0) moves a shape:	**50 units to the right (no vertical change)** — Translating by (50, 0) shifts the shape 50 units along x (right) and 0 along y (no vertical move). The translation vector's components control horizontal and vertical movement separately.
PREDICT: a shape is scaled by 2, then by 0.5. The net effect is:	**No change in size (2 x 0.5 = 1)** — Successive scales multiply: scaling by 2 then by 0.5 gives a net factor of 1 (2 x 0.5) — no overall size change. Transforms compose by combining their effects.
PREDICT: scaling a shape by a factor of 0.5 makes it:	**Half as large** — A scale factor of 0.5 halves the shape's size. Factors between 0 and 1 shrink a shape; factors above 1 grow it.
'The order of transformations doesn't affect the result.' This is:	**False — transform order matters (rotate-then-move differs from move-then-rotate)** — MISCONCEPTION NAMED — 'order doesn't matter.' Transformation order affects the result: rotating then translating places a shape differently than translating then rotating. Order is a crucial, often-surprising factor in transforms.
PREDICT: rotating a square by 90° four times returns it to:	**Its original orientation (4 x 90° = 360°)** — Four 90° rotations total 360° — a full turn — so the square returns to its original orientation. Rotations accumulate; 360° brings a shape back to where it started.
Animating a transform over time (e.g., increasing rotation each frame) creates:	**Motion (a spinning, growing, or moving shape)** — Animating a transformation — increasing rotation, position, or scale each frame — produces motion: a spinning, moving, or growing shape. Combining transforms with changing state drives animation.
To rotate a shape around its OWN center (not the origin), you typically:	**Translate to the center, rotate, then draw** — To rotate around a shape's own center, you translate the origin to that center, then rotate, then draw — otherwise the shape rotates around the canvas origin (0,0), swinging in a large arc instead of spinning in place.
PREDICT: a shape is translated to (100, 100), then a circle is drawn at (0, 0). It appears at:	**(100, 100) — the translation shifted the coordinate origin** — After translating to (100,100), drawing at (0,0) places the circle at (100,100) — translation shifts the coordinate origin, so subsequent draws are relative to the new origin. A key transform behavior.
PREDICT: drawing a shape, rotating a bit, and repeating in a loop creates:	**A radial/spiral pattern** — Drawing a shape, rotating slightly, and repeating in a loop fans copies out radially — creating a radial or spiral pattern (like flower petals or a pinwheel). Looped rotation is a staple generative technique.
Rotation in code is usually measured in:	**Degrees or radians (an angle)** — Rotation is specified as an angle, in degrees or radians (many coding environments use radians by default). The angle determines how far the shape turns around the pivot.
Transformations are	**One shape's transform doesn't unintentionally affect others** — Push/pop

often 'saved and restored' (push/pop) so that:	(save/restore) isolate transformations so that translating/rotating for one shape doesn't carry over and misplace subsequent shapes — a key technique for managing transform state cleanly.
ROTATE means to:	**Turn a shape around a point by an angle** — Rotate turns a shape around a pivot point by a given angle — one of the core transformations, used to spin or angle elements.
PREDICT: scaling by a factor of 1 leaves a shape:	**Unchanged in size** — A scale factor of 1 multiplies dimensions by 1 — leaving the shape unchanged in size. Factor 1 is the 'identity' scale (no effect).
TRANSLATE means to:	**Move a shape to a new position** — Translate means to move a shape to a new position (shifting it along x and/or y). It's one of the three basic transformations (translate, rotate, scale).
Understanding shape and transforms lets a creative coder:	**Position, orient, size, and animate shapes to build and move compositions** — Understanding translate, rotate, and scale (and their order) lets a creative coder position, orient, size, and animate shapes — building complex, moving compositions from simple forms and predicting how transforms combine.
PREDICT: scaling a shape by a factor of 2 makes it:	**Twice as large** — A scale factor of 2 doubles the shape's size in each dimension (twice as wide and tall). Scale factors multiply the shape's dimensions.
Combining translate + rotate + scale lets a coder:	**Place, orient, and size shapes flexibly to build compositions** — Combining translate (position), rotate (orientation), and scale (size) gives full flexible control to place, orient, and size shapes anywhere — the toolkit for building complex compositions from simple shapes.
SCALING a shape non-uniformly (different x and y factors) will:	**Stretch/distort it (e.g., a circle becomes an ellipse)** — Scaling with different x and y factors stretches/distorts a shape — a circle scaled 2x horizontally and 1x vertically becomes a wide ellipse. Non-uniform scaling changes proportions.
SCALE means to:	**Change a shape's size** — Scale changes a shape's size — a factor greater than 1 enlarges it, less than 1 shrinks it, and 1 leaves it unchanged. It's the size transformation.

7. Arranging generated elements in space

✂ Question (fold →)	Answer
A RULE OF THIRDS-style placement in code would put a focal element:	**Off-center, around 1/3 across the canvas** — Applying the rule of thirds, a focal element is placed off-center, around one-third across the canvas (e.g., x = width/3) — often more dynamic and pleasing than dead-center placement.
Which best creates an EVENLY-SPACED ROW of 6 circles across a 600-wide canvas?	**A loop placing circle i at x = i x 100** — A loop placing circle i at x = i x 100 (with 600/6 = 100 spacing) creates an evenly-spaced row of 6 circles across the canvas. Index x spacing is the standard even-layout formula.
Negative space (empty areas) in a composition:	**Is important — it gives the eye rest and defines the elements** — Negative (empty) space is important in generative composition too — it gives the eye rest, creates focus, and defines the positive elements. Filling every pixel usually creates clutter, not richness.
PREDICT: elements placed at angle = i x (360/8) around a center generate:	**8 elements evenly arranged in a circle** — Placing 8 elements at angles of i x 45° (360/8) around a center point arranges them evenly in a circle — a radial layout, using trigonometry (or looped rotation) to position elements around a center.
Scaling a composition to different canvas sizes is easier if positions are:	**Computed relative to width/height (proportional), not hardcoded** — Computing positions proportionally (e.g., x = width x 0.25) rather than hardcoding pixels lets a composition scale gracefully to different canvas sizes — a responsive-layout principle in code.
'To center an element, place it at (0,0).' This is:	**False — center is usually (width/2, height/2), not the origin** — MISCONCEPTION NAMED — '(0,0) is the center.' On a typical canvas, (0,0) is the top-left corner; the CENTER is (width/2, height/2). Placing an element at (0,0) puts it in the corner, not the middle.
Using the loop index to compute position lets a coder:	**Arrange elements in structured layouts (rows, grids, circles)** — Using the loop index in a position formula (e.g., x = i x spacing) arranges elements in structured layouts — rows, grids, or circles — turning repetition into intentional composition.
Combining a grid with per-cell variation (color, size, rotation) creates:	**A structured yet varied composition** — Combining a grid's structure with per-cell variation (varying each cell's color, size, or rotation) creates a composition that's organized yet visually rich — a hallmark of much generative art.
To place text or a focal shape with breathing room, a coder should:	**Leave margins/negative space around it** — Leaving margins and negative space around a focal element gives it breathing room and prominence. Crowding elements against edges or over each other reduces clarity and impact.
Layering (drawing order) matters because:	**Later-drawn elements appear ON TOP of earlier ones** — Draw order determines layering: elements drawn later appear on top of earlier ones. So you typically draw the background first and foreground elements last — controlling depth through order.
PREDICT: mirroring elements across the canvas's vertical center line creates:	**A symmetrical composition** — Mirroring elements across the vertical center line creates a symmetrical composition — for each element at x, draw a partner at (width - x). Symmetry is a strong, balanced compositional strategy.
PREDICT: adding a small random offset	**A grid that looks slightly organic/hand-placed** — Adding a small random offset (jitter)

(jitter) to each element's grid position creates:	to each grid position loosens the rigid grid into a slightly organic, hand-placed look — combining structure (grid) with controlled randomness for a natural feel.
PREDICT: to spread 5 elements evenly across a canvas of width 500, you'd space them about:	**100 units apart** — To spread 5 elements evenly across a 500-wide canvas, space them roughly width/count = 500/5 = 100 units apart. Dividing the space by the number of elements gives even spacing.
VISUAL BALANCE in a generated composition means:	**Distributing visual weight so it doesn't feel lopsided** — Visual balance distributes the composition's visual weight (size, density, color) so it doesn't feel lopsided — achievable through symmetry or considered asymmetric placement even in generative work.
PREDICT: a loop places rectangles whose HEIGHT = i x 10 in a row. The result is:	**Rectangles of increasing height (like a bar chart / staircase)** — Setting each rectangle's height to i x 10 makes them grow taller across the row — a staircase or bar-chart look. Using the loop index for size (not just position) creates structured variation.
Understanding composition and layout lets a creative coder:	**Arrange generated elements into structured, balanced, intentional designs** — Understanding composition and layout — grids, radial arrangements, spacing, balance, negative space, and draw order — lets a creative coder arrange generated elements into structured, balanced, intentional designs rather than random clutter.
The canvas (drawing area) has a coordinate system where (0,0) is usually:	**The top-left corner** — In most coding canvases, (0,0) is the top-left corner, with x increasing rightward and y increasing DOWNWARD. Knowing the origin is essential for positioning elements correctly.
PREDICT: nested loops draw a dot at (col x 30, row x 30) for cols 0-4 and rows 0-4. The result is:	**A 5x5 grid of 25 dots** — Looping cols 0-4 and rows 0-4, placing dots at (col x 30, row x 30), produces a 5x5 grid of 25 evenly-spaced dots. Multiplying the loop index by a spacing value positions grid elements.
A GRID layout in code is generated by:	**Nested loops placing elements at row/column positions** — A grid layout is generated by nested loops: the outer loop for rows, the inner for columns, placing an element at each (row, column) position with regular spacing — a fundamental compositional structure.
PREDICT: drawing a large background rectangle AFTER the shapes will:	**Cover the shapes (they're hidden behind it)** — Drawing a filled background rectangle AFTER the shapes covers them — since the last-drawn element is on top. To keep shapes visible, draw the background FIRST. Draw order is a common source of 'where did my shapes go?' bugs.

8. Timing musical events programmatically

✂ Question (fold →)	Answer
PREDICT: a pattern plays hits on beats 1 and 3 of a 4-beat measure. This creates:	**A sparse, on-the-beat rhythm (every other beat)** — Playing hits only on beats 1 and 3 of a 4-beat measure creates a sparse rhythm sounding every other beat — a steady, on-the-beat feel (common for a bass drum). Which beats have events defines the rhythm.
PREDICT: a 16-step sequencer with hits on steps 1, 5, 9, 13 produces:	**A steady four-on-the-floor pulse (every 4th step)** — Hits on steps 1, 5, 9, 13 of 16 (every 4th step) produce 4 evenly-spaced pulses — a steady 'four-on-the-floor' beat. Even spacing in the step grid yields a regular rhythm.
PREDICT: shifting a rhythmic pattern to start one step later (offset) creates:	**A displaced groove (the same pattern, shifted in time)** — Offsetting a pattern to start one step later shifts the entire groove in time — the same rhythm, displaced. Rhythmic offsets create variation and can produce interesting phase/polyrhythmic effects.
A POLYRHYTHM is created by:	**Playing two rhythms with different groupings at once (e.g., 3 against 4)** — A polyrhythm layers two rhythms with different groupings simultaneously (e.g., 3 evenly-spaced hits against 4) — the patterns align and diverge over time, creating complex, interlocking rhythms.
Tempo is independent of the NUMBER of notes because:	**Tempo sets beat speed; how many notes fit depends on subdivision** — Tempo (beat speed) is independent of note density: at a fixed tempo you can play few notes (whole beats) or many (fine subdivisions). Adding more notes via subdivision doesn't change the tempo — it fills the beats more densely.
A BEAT is:	**The basic unit of time in music (a steady pulse)** — A beat is the basic, steady pulse of music — the unit you'd tap your foot to. Rhythmic events are timed relative to beats, and tempo sets how fast beats occur.
Doubling the tempo of a pattern makes it:	**Play twice as fast (half the duration)** — Doubling the tempo (BPM) plays the pattern twice as fast, so it takes half the time. Tempo scales the timing of all events proportionally.
SYNCOPATION in code is created by:	**Placing hits on off-beats (between the main beats)** — Syncopation is created by placing rhythmic events on off-beats (the 'and' between main beats) rather than on the strong beats — producing a groove with unexpected emphasis and drive.
PREDICT: increasing the tempo (BPM) value makes the music:	**Play faster** — Increasing BPM increases beats per minute, so the music plays faster. Tempo controls speed, independent of pitch (higher/lower) or volume (louder/softer).
Which best creates a steady 4-beat drum loop at 100 BPM?	**A loop scheduling a kick on each of 4 beats, timed to 100 BPM** — A loop scheduling a kick on each of the 4 beats, timed to 100 BPM, creates a steady 4-beat drum loop. Playing them all at once (no timing) or randomly would not produce a steady rhythm.
TEMPO in music code is:	**The speed of the beat (e.g., in beats per minute)** — Tempo is the speed of the beat, usually measured in beats per minute (BPM). Higher BPM means a faster piece; changing the tempo value speeds up or slows down the music.
PREDICT: a loop plays a note every beat for 4 beats at 120	**2 seconds (4 beats / 120 BPM x 60)** — At 120 BPM, there are 2 beats per second (120/60), so 4 beats take 2 seconds. Converting beats to real time uses the tempo:

BPM. It lasts:	seconds = beats / (BPM/60).
Subdividing a beat into smaller units (e.g., eighth notes) lets you:	**Play faster/more intricate rhythms** — Subdividing a beat (into eighth notes, sixteenth notes, etc.) lets you place events between the main beats — creating faster, more intricate rhythms within the same tempo.
In code, timing musical events accurately requires:	**Scheduling events according to the tempo/beat clock** — Accurate musical timing in code requires scheduling events according to a tempo-based beat clock — computing WHEN each note/hit should sound based on the tempo, rather than firing them instantly or randomly.
A STEP SEQUENCER represents rhythm as:	**A grid of on/off steps (each step = a time slot)** — A step sequencer represents rhythm as a grid of on/off steps — each step is a time slot, and toggling it on plays a sound there. Looping through the steps generates a repeating rhythmic pattern.
PREDICT: a rhythmic loop of 4 beats is repeated 4 times. The total is:	**16 beats** — Repeating a 4-beat loop 4 times gives 16 beats total — loops accumulate their length. This is how a short rhythmic pattern builds into a longer section.
PREDICT: a hi-hat plays on every subdivision (16 sixteenth-notes per bar) while a kick plays on beats 1 and 3. Together they create:	**A layered rhythm — steady hi-hats over a sparser kick** — A hi-hat on every subdivision over a kick on beats 1 and 3 layers two rhythmic patterns — a steady, dense hi-hat with a sparser kick foundation — building a fuller groove. Layering patterns creates rhythmic texture.
Understanding rhythm and tempo in code lets a creative coder:	**Programmatically time musical events into grooves, patterns, and rhythms** — Understanding rhythm and tempo in code — beats, tempo/BPM, subdivision, sequencing, syncopation, and layering — lets a creative coder programmatically time musical events into grooves and patterns, predicting how timing choices sound.
'Changing the tempo also changes the pitch of the notes.' In code (event timing), this is:	**False — tempo affects timing/speed, not pitch** — MISCONCEPTION NAMED — 'tempo changes pitch.' In code that sequences musical events, tempo controls the SPEED/timing of events; pitch is a separate parameter. Speeding up the tempo plays the same notes faster, not higher.
A REST in a rhythm pattern represents:	**A silent time slot (no sound)** — A rest is a silent time slot — in a step sequencer it's an 'off' step. Rests are as important as hits: the pattern of sound AND silence defines the rhythm.

9. Choosing pitches programmatically

✂ Question (fold →)	Answer
An INTERVAL in code is:	**The distance in pitch between two notes** — An interval is the pitch distance between two notes, measured in half steps (e.g., 7 half steps = a perfect fifth). Intervals are the building blocks of scales and chords in code.
HARMONY in code means:	**Combining pitches (chords) to support a melody** — Harmony is the combination of pitches (chords) sounding together, typically supporting a melody. In code, generating harmony means triggering chord tones alongside the melodic notes.
PREDICT: two notes 7 half steps apart form a:	**Perfect fifth** — Two notes 7 half steps apart form a perfect fifth (e.g., C to G, MIDI 60 to 67) — a consonant, stable interval. Counting half steps identifies the interval.
A MIDI note number represents:	**A specific pitch as an integer (e.g., 60 = middle C)** — MIDI note numbers represent pitches as integers (60 = middle C); adding 1 raises the pitch a half step, adding 12 raises it an octave. This makes pitch easy to compute in code.
Which approach best generates a MUSICAL random melody?	**Pick random notes from a chosen scale** — Picking random notes from a chosen scale generates a musical random melody — varied yet coherent. Picking any random frequency sounds dissonant; repeating one note isn't a melody; random tempo affects timing, not pitch.
A MINOR scale in code will generally produce melodies that sound:	**Darker or sadder than major** — Constraining notes to a minor scale generally produces darker or sadder-sounding melodies than a major scale — the different pattern of intervals gives minor its characteristic mood.
PREDICT: adding 12 to a MIDI note number raises the pitch by:	**One octave** — Adding 12 to a MIDI note number raises the pitch by 12 half steps = one octave (e.g., 60 -> 72, middle C to the C above). Octave arithmetic is simply +/-12.
PREDICT: a melody generated from a pentatonic scale tends to sound:	**Consonant and pleasant (hard to hit 'wrong' notes)** — A pentatonic (5-note) scale tends to produce consonant, pleasant melodies because it omits the notes most likely to clash — it's 'forgiving,' so even random choices within it sound musical. A popular choice for generative music.
Playing the same melody starting on a different pitch (all intervals preserved) is:	**Transposition** — Transposition shifts every pitch by the same interval, preserving the melody's shape (its intervals) while moving it to a new starting pitch/key — done in code by adding a constant to every note number.
To transpose a melody UP a whole step in code, you would:	**Add 2 to each note's pitch (2 half steps)** — Transposing up a whole step means raising each pitch by 2 half steps — add 2 to each MIDI note number. Transposition in code is arithmetic on pitch values (add/subtract half steps).
To keep a generative melody in KEY, a coder should:	**Restrict notes to that key's scale** — To keep a melody in key, restrict (or 'snap') note choices to the key's scale — filtering out off-key pitches. This ensures generated melodies stay harmonically coherent within the chosen key.

'Any random pitches will sound musical.' This is:	**False — random pitches usually sound dissonant; constraining to a scale keeps them musical** — MISCONCEPTION NAMED — 'random pitches sound musical.' Fully random pitches usually sound dissonant/chaotic. Constraining choices to a scale (or chord) is what keeps generated melodies musical and coherent.
A SCALE in code is often represented as:	**A list/set of pitches to choose from** — A scale is represented as a list/set of allowed pitches (e.g., the C major scale). Restricting note choices to a scale ensures generated melodies sound musically coherent.
Mapping a data value to a pitch (e.g., temperature -> note) is called:	**Sonification (turning data into sound)** — Mapping data to pitch (higher data = higher note) is sonification — representing data as sound. It lets you 'hear' patterns in data, a creative and analytical use of code-generated pitch.
PREDICT: playing MIDI notes 60, 64, and 67 together produces:	**A C major chord (root, third, fifth)** — MIDI 60, 64, 67 are C, E, G (a major third + a minor third apart) — a C major chord. Playing them simultaneously sounds the chord. Chords are built by stacking intervals of pitch.
Understanding pitch, scale, and harmony in code lets a creative coder:	**Generate coherent melodies and chords by constraining pitch choices** — Understanding pitch (as numbers), scales (constrained pitch sets), intervals, and harmony (chords) lets a creative coder generate coherent melodies and chords — constraining pitch choices so algorithmic music sounds musical.
PITCH in music code refers to:	**How high or low a note sounds** — Pitch is how high or low a note sounds (its frequency). In code, pitch is a parameter you set for each note — separate from duration (length), volume (loudness), and tempo (speed).
PREDICT: an ARPEGGIO plays a chord's notes:	**One after another (in sequence) rather than together** — An arpeggio plays a chord's notes one after another in sequence (a 'broken chord') rather than simultaneously — e.g., playing C, then E, then G. It's a melodic way to express a chord.
A CHORD in code is:	**Multiple pitches played at the same time** — A chord is multiple pitches played simultaneously (e.g., C, E, G together for a C major chord). In code, you trigger several notes at the same time to sound a chord.
PREDICT: choosing random notes ONLY from a major scale produces:	**Melodies that sound coherent/consonant** — Choosing random notes restricted to a scale produces coherent, consonant-sounding melodies — the scale constrains the randomness so results stay musical, unlike picking any pitch at random (which sounds chaotic).

10. Building and transforming melodic ideas in code

✂ Question (fold →)	Answer
PREDICT: AUGMENTING a motif (doubling each note's duration) makes it:	**Play slower/more stretched out (same pitches)** — Augmentation doubles each note's duration, stretching the motif out in time while keeping the same pitches — a slower, grander version. (Diminution, the opposite, shortens durations for a faster version.)
A MELODY is:	**A sequence of single pitches heard as a tune** — A melody is a sequence of single pitches played one after another, heard as a coherent tune. In code, a melody is a list of notes (pitch + duration) played in order.
A melody built mostly by STEPWISE motion (adjacent scale notes) sounds:	**Smooth and singable** — A melody moving mostly by steps (adjacent scale notes) sounds smooth and singable. Melodies with large leaps sound more angular/dramatic. Contour and interval size shape a melody's character.
PREDICT: transposing a motif so it fits a new chord (staying in key) produces:	**A version of the motif that harmonizes with the new chord** — Transposing/adjusting a motif to fit a new chord (while staying in key) produces a version that harmonizes with that chord — reusing the melodic idea over changing harmony, a common way to develop a theme.
'A good melody uses each note only once and never repeats ideas.' This is:	**False — repetition and recurring motifs are central to memorable melodies** — MISCONCEPTION NAMED — 'never repeat.' Repetition and recurring motifs are central to memorable, coherent melodies. Good melodies balance repetition (for memorability) with variation (for interest) — not endless novelty.
PREDICT: TRANSPOSING a motif up by 5 half steps (adding 5 to each pitch) produces:	**The same melodic shape, higher in pitch** — Transposing a motif up 5 half steps (adding 5 to each note) preserves its melodic SHAPE (intervals) while moving it higher — a recognizable variation of the same idea. A core motif-development technique.
INVERTING a motif means:	**Flipping its melodic direction (ups become downs)** — Inverting a motif flips its melodic direction — where it rose, it now falls by the same intervals, and vice versa. It's a classic transformation that varies a motif while keeping it recognizable.
A melody's RHYTHM (note durations) is:	**As important as its pitches to its identity** — A melody's rhythm (the pattern of note durations) is as important as its pitches — the same pitches with a different rhythm produce a very different melody. Melody = pitch sequence + rhythm together.
PREDICT: playing a motif, then the SAME motif a step lower, then a step lower again, is a:	**Descending sequence** — A motif repeated at successively lower pitch levels is a descending sequence — the idea 'walks' downward. Sequences (ascending or descending) are a clear, effective motif-development device.
Which best describes MOTIF DEVELOPMENT in code?	**Reusing a short idea with transformations (transpose, invert, retrograde, augment)** — Motif development reuses a short core idea with transformations — transposition, inversion, retrograde, augmentation, sequences — creating coherent variety. This is how a memorable idea becomes a whole piece.
PREDICT: reversing the note list [C, D, E, F]	**[F, E, D, C]** — Reversing the list [C, D, E, F] gives [F, E, D, C] — the retrograde. In code, transforming a melody often means manipulating its note list (reversing, shifting,

gives:	mapping).
A MOTIF is:	**A short, memorable musical idea that recurs** — A motif is a short, distinctive musical idea (a few notes) that recurs and is developed throughout a piece — a memorable building block. In code, you define a motif and reuse/transform it.
In code, a melody can be stored as:	**A list of notes (each with pitch and duration)** — A melody is stored as an ordered list of notes, each with a pitch and duration. Playing through the list in order performs the melody — and manipulating the list transforms it.
MELODIC CONTOUR refers to:	**The overall shape of a melody's rises and falls** — Melodic contour is the overall shape of a melody's rises and falls — its outline over time. Two melodies can share a contour (shape) even at different pitches; contour is a key element of melodic identity.
PREDICT: taking a motif and playing it, then its inversion, then its retrograde generates:	**Three related variations of one idea** — Playing a motif, then its inversion, then its retrograde generates three related variations of one core idea — coherent development that gives a piece unity (recognizable) and variety (transformed). A composer's toolkit in code.
RETROGRADE of a motif means:	**Playing it backward (last note first)** — Retrograde plays a motif backward — reversing the order of its notes (the last note becomes first). In code, this is reversing the note list. It's another transformation for developing a motif.
Combining a repeating motif with slight variations each time achieves:	**Unity (recognizable) plus interest (varied)** — Combining a recurring motif with slight variations each repetition achieves the balance of unity (the idea stays recognizable) and variety (it stays interesting) — the essence of compelling melodic development.
Repeating a motif at a HIGHER pitch each time creates a:	**Sequence (the motif climbing/descending in steps)** — Repeating a motif at successively higher (or lower) pitch levels creates a musical sequence — the same idea climbing or descending, a powerful way to develop a motif and build momentum.
Understanding melody and motif lets a creative coder:	**Generate and develop coherent musical ideas through transformation** — Understanding melody and motif lets a creative coder generate and develop coherent musical ideas — defining a motif and transforming it (transpose, invert, retrograde, sequence) to build melodies with unity and variety.
Generating a melody by picking scale notes with a bias toward SMALL steps produces:	**A smooth, coherent, singable melody** — Biasing note choices toward small steps (adjacent scale notes) generates smoother, more coherent, singable melodies — controlling not just WHICH notes (the scale) but HOW they connect (the intervals) improves algorithmic melody quality.

11. Building rich visual and sonic layers

✂ Question (fold →)	Answer
Understanding texture and layering lets a creative coder:	**Build rich, deep compositions by combining well-differentiated layers** — Understanding texture and layering lets a creative coder build rich, deep visual and sonic compositions by combining well-differentiated layers — balancing fullness with clarity, and predicting how layers combine.
VISUAL TEXTURE (roughness/pattern) can be generated by:	**Many small repeated marks (dots, lines, strokes)** — Visual texture (roughness, pattern) is generated by many small repeated marks — dots (stippling), lines (hatching), or strokes — accumulating into a textured surface. Density and variation control the texture's feel.
PREDICT: layering a bass line, a chord pad, AND a melody creates a texture that is:	**Fuller/thicker than any single part alone** — Layering a bass, chords, and a melody creates a fuller, thicker texture than any single part alone — the combined layers produce a rich, complete-sounding arrangement. Each layer adds a role.
PREDICT: removing all but one layer from a rich composition leaves it:	**Thinner/sparser (less full)** — Removing all but one layer leaves a composition thinner and sparser — losing the fullness the layers provided. This shows how much layering contributes to a rich texture (and how a sparse texture is a valid, deliberate choice).
POLYPHONIC texture in music is:	**Multiple independent melodic lines sounding together** — Polyphonic texture layers multiple independent melodic lines simultaneously (like a round or fugue) — a rich, complex texture where each layer is its own melody, interweaving with the others.
Using TRANSPARENCY (low alpha) when layering shapes creates:	**Soft blending and color buildup between layers** — Low-alpha (transparent) layers blend where they overlap, building soft color and a sense of depth — a key technique for rich, atmospheric generative textures rather than flat, opaque stacking.
MONOPHONIC texture in music is:	**A single melodic line with no accompaniment** — Monophonic texture is a single melodic line with no accompaniment — the thinnest texture. Layering adds accompaniment (homophony) or independent lines (polyphony) for thicker textures.
PREDICT: adding a subtle background texture layer BEHIND the main shapes:	**Adds depth without distracting from the focal elements** — A subtle background texture layer behind the main shapes adds depth and atmosphere without distracting — supporting the focal elements. Layering with a clear hierarchy (subtle back, prominent front) enriches without cluttering.
LAYERING in generative art means:	**Drawing elements on top of each other to build depth/richness** — Layering draws elements on top of one another to build depth and visual richness — e.g., a background, then midground shapes, then foreground details, each layer adding complexity.
A HOMOPHONIC music texture is:	**A main melody supported by chords/accompaniment** — Homophonic texture is a main melody with supporting chordal accompaniment (like most songs) — a layered texture where one layer leads (melody) and others support (harmony). It sits between monophony and polyphony.
PREDICT: drawing thousands of tiny semi-transparent dots with slight random variation creates:	**A soft, organic textured surface** — Thousands of tiny, semi-transparent, slightly-varied dots accumulate into a soft, organic textured surface — a generative stippling technique where controlled repetition + transparency + randomness build rich texture.

Varying the layers' opacity/volume creates:	**A sense of depth and hierarchy among them** — Varying opacity (visual) or volume (sonic) among layers creates depth and hierarchy — more prominent layers come forward, subtler ones recede. Managing layer prominence organizes a rich composition.
Each layer in a composition should ideally:	**Have a distinct role (foreground/background, bass/melody)** — Each layer should have a distinct role — visually (background vs focal point) or sonically (bass vs harmony vs melody) — so layers complement rather than compete, keeping the texture rich but clear.
In music, a DRONE layer is:	**A sustained continuous tone underneath other parts** — A drone is a sustained, continuous tone (or chord) held underneath other musical layers — a foundational texture layer that grounds the harmony while melody and rhythm move above it.
To keep a layered composition CLEAR (not muddy), a coder should:	**Give layers distinct roles, frequencies/positions, and prominence** — To keep layering clear, give each layer a distinct role, position/frequency range, and level of prominence — so layers occupy different 'space' and complement each other, avoiding a muddy, cluttered result.
PREDICT: layering multiple slightly-offset copies of the same shape (with transparency) creates:	**A blurred, motion-like or glowing effect** — Layering slightly-offset, transparent copies of a shape creates a blurred, glowing, or motion-trail effect — the overlapping copies accumulate into a soft, dynamic-looking form. A common generative effect.
In MUSIC, texture refers to:	**How many layers/voices sound together and their relationship** — Musical texture describes how many layers/voices sound together and how they relate — thin (a single melody) to thick (many simultaneous parts). Layering instruments/parts builds texture.
Which best builds a rich but CLEAR sonic texture?	**A bass, a chord layer, and a melody — each in its own register and role** — A bass, chord layer, and melody — each occupying its own register and role — builds a rich yet clear texture. Ten melodies in the same range would clash into mud; a single note is thin. Distinct roles/registers keep layers clear.
'More layers always make better art/music.' This is:	**False — too many layers can create clutter/muddiness; balance matters** — MISCONCEPTION NAMED — 'more layers = better.' Too many visual layers create clutter; too many sonic layers create mud. Effective texture balances layers, giving each room and purpose — quality and clarity over sheer quantity.
In both art and music, TEXTURE contributes:	**Richness, depth, and interest** — Texture (built through layering) contributes richness, depth, and interest in both visual and musical work — the interplay of layers turns a simple idea into a full, engaging composition.

12. Responding to input in real time

✂ Question (fold →)	Answer
Mapping the MOUSE POSITION to a parameter (e.g., size) lets a user:	**Control the artwork in real time by moving the mouse** — Mapping mouse position to a parameter (e.g., mouseX -> circle size) gives the user real-time control — moving the mouse continuously changes the output. Input-driven parameters create interactive art.
Understanding interaction and events lets a creative coder:	**Build responsive art and music that users shape in real time** — Understanding events, the draw loop, conditionals, state, and input mapping lets a creative coder build responsive, interactive art and music that users shape in real time — predicting how a program reacts to input.
An EVENT HANDLER is:	**Code that runs when a specific event occurs** — An event handler is code that runs in response to a specific event — e.g., an onClick handler runs when the mouse is clicked. It defines the program's reaction to that event.
Which best describes an INTERACTIVE program?	**One that continuously responds to user input in real time** — An interactive program continuously responds to user input in real time (via events and a draw loop). A one-time fixed drawing, or a program ignoring input, is not interactive.
Well-designed interaction gives the user:	**Clear, responsive feedback for their actions** — Well-designed interaction provides clear, immediate feedback — the user's action produces a visible/audible reaction — so they understand how their input affects the piece. Responsiveness makes interaction satisfying and legible.
PREDICT: 'when the mouse is clicked, draw a circle at the cursor' — clicking 3 times produces:	**3 circles at the 3 click locations** — The click handler runs once per click, drawing a circle at each cursor location — 3 clicks produce 3 circles at the 3 spots. Event handlers respond each time their event fires.
PREDICT: each click TOGGLES a boolean 'isOn' (true<->false) that controls whether music plays. Two clicks return it to:	**Its original state (toggled twice)** — Toggling a boolean twice returns it to its original value (true->false->true). A toggle flips state each event; an even number of toggles cancels out. Combining events with a state variable enables on/off controls.
An event that fires REPEATEDLY while a key is held (vs once on press) would:	**Continuously trigger its action as long as the key is down** — An event (or a per-frame check) that responds while a key is HELD continuously triggers its action as long as the key is down — e.g., moving a shape as long as an arrow key is pressed. This differs from a single press event.
STATE that changes on interaction (e.g., a toggle) requires:	**A variable to remember the current state** — Interaction that changes state (like a toggle switched by a click) requires a variable to remember the current state (on/off) between events — combining event handling with state to produce persistent, responsive behavior.
PREDICT: 'if mouseX > width/2, background is blue, else red' — moving the mouse across the canvas:	**Switches the background between red and blue at the midpoint** — With that conditional, the background is red when the mouse is on the left half and blue on the right — switching at the midpoint as the mouse crosses. The conditional maps input (position) to different outputs.
	Something that happens (like a mouse click) that code can respond to — An event is something that happens — a mouse click, a key press, a timer tick — that

An EVENT in code is:	code can respond to. Event-driven code reacts to these triggers, enabling interactivity.
Interactive art differs from static generative art in that it:	**Responds to the user's ongoing input** — Interactive art responds to the user's ongoing input (mouse, keys, sensors), so the user helps shape the output in real time — unlike static generative art, which produces a fixed result without live interaction.
CONDITIONALS (if-statements) in interactive code let a program:	**Do different things based on the input/state** — Conditionals (if-statements) let interactive code branch — doing different things depending on the input or state (e.g., 'if the mouse is on the left half, do X; else Y'). They give interactivity its decision-making.
'Interactive programs run their whole output once and then stop.' This is:	**False — interactive programs run continuously, responding to ongoing input** — MISCONCEPTION NAMED — 'run once and stop.' Interactive programs run continuously (often in a draw loop), responding to ongoing input (mouse, keys) — the output evolves as the user interacts, rather than producing one fixed result.
A KEY-PRESS event lets a program:	**React when the user presses a keyboard key** — A key-press event lets a program react to keyboard input — e.g., pressing the spacebar could trigger a sound or change the visuals. Keyboard events are a common interactivity source.
The DRAW LOOP (a function that runs every frame) is used to:	**Continuously update and redraw for animation/interaction** — The draw loop runs repeatedly (often ~60 times/second), updating state and redrawing each frame — enabling continuous animation and real-time response to input. It's the heartbeat of interactive/animated programs.
Mapping input to MULTIPLE parameters (e.g., mouseX->size, mouseY->color) creates:	**Rich, multidimensional interactive control** — Mapping mouseX to one parameter (size) and mouseY to another (color) gives rich, two-dimensional interactive control — the user shapes multiple aspects at once by moving the mouse. Multi-parameter mapping deepens interactivity.
PREDICT: a program plays a note whenever a key is pressed. Pressing keys quickly produces:	**A sequence of notes played in time with the presses** — A program playing a note per keypress turns the keyboard into an instrument — pressing keys quickly plays a sequence of notes timed to the presses. Event handlers make real-time musical interaction possible.
PREDICT: in the draw loop, a shape's x-position is set to the mouse's x. The shape will:	**Follow the mouse horizontally in real time** — Setting the shape's x to mouseX every frame in the draw loop makes it follow the cursor horizontally in real time — the loop continuously reads the current input and updates the output.
PREDICT: dragging the mouse while it's pressed leaves a trail of shapes. This uses:	**A 'mouse dragged' event (or checking mouse-pressed each frame)** — Leaving a trail while dragging uses a mouse-dragged event (or checking 'is the mouse pressed?' each frame in the draw loop) and drawing at the cursor — producing a continuous trail as the user drags. A classic interactive-drawing behavior.

13. Reflection, rotation, and repeating patterns

✂ Question (fold →)	Answer
'A design can only have ONE type of symmetry.' This is:	**False — a design can combine reflection AND rotational symmetry** — MISCONCEPTION NAMED — 'only one symmetry type.' A design can have multiple symmetries at once — a snowflake has both rotational (6-fold) AND reflection symmetry. Combining symmetries creates richer patterns.
Which best generates a REPEATING TILED pattern?	**A nested grid loop placing a tile at each cell** — A nested grid loop placing a tile at each cell generates a repeating tiled pattern (translational symmetry). One tile alone isn't repeated; random scattering isn't tiled; a single rotation gives rotational, not tiled, symmetry.
PREDICT: a tile that connects seamlessly at its edges, repeated in a grid, produces:	**A continuous pattern with no visible seams** — A tile designed so its edges match its neighbors, repeated in a grid, produces a continuous, seamless pattern (no visible seams) — the goal of tileable/seamless texture design in generative art.
PREDICT: rotating a motif by 45° repeatedly (8 times) gives ___-fold symmetry:	**8-fold (360° / 45° = 8)** — Rotating by 45° eight times (8 x 45° = 360°) places 8 copies around a full turn — 8-fold rotational symmetry. The order equals 360° divided by the rotation angle.
PREDICT: a snowflake generator reflects a random branch into 6 rotated segments. Each snowflake is:	**Symmetric (6-fold) but unique due to the random branch** — Reflecting a randomly-generated branch into 6 rotated segments makes each snowflake 6-fold symmetric yet unique (the random branch differs each time) — combining symmetry (structure) with randomness (variety), like real snowflakes.
PREDICT: repeating a motif rotated by 60° six times (6 x 60° = 360°) creates:	**A design with 6-fold rotational symmetry (like a snowflake)** — Repeating a motif rotated by 60° six times fills a full circle (360°) with 6 evenly-arranged copies — a design with 6-fold rotational symmetry, like a snowflake. Looped rotation generates rotational symmetry.
TILING (tessellation) means:	**Covering a plane with repeated shapes and no gaps** — Tiling (tessellation) covers a plane with repeated shapes leaving no gaps or overlaps — like floor tiles. In code, a grid loop placing a repeating tile generates a tessellation.
A TRANSLATION-based tiling repeats a tile by:	**Shifting it by a fixed offset across the plane** — Translational symmetry (the basis of grid tiling) repeats a tile by shifting it a fixed offset horizontally and vertically — copies translated across the plane fill it without rotation or reflection.
Symmetry in code is efficiently created using:	**Transformations (reflect, rotate) in a loop** — Symmetry is efficiently generated by using transformations — reflecting and rotating a base element in a loop — so the code computes the symmetric copies rather than the artist placing each by hand.
Understanding symmetry and tiling lets a creative coder:	**Generate ordered, balanced, and intricate repeating patterns** — Understanding symmetry (reflection, rotational) and tiling (tessellation) lets a creative coder generate ordered, balanced, and intricate repeating patterns using transformations — predicting the symmetric structure code will produce.
PREDICT: mirroring a pattern both HORIZONTALLY and VERTICALLY creates:	**4-way (quadrant) symmetry** — Mirroring a pattern across both a horizontal and a vertical axis produces 4-way (quadrant) symmetry — the design is reflected into four matching quadrants. Combining two reflections multiplies the symmetry.

Islamic geometric art and many natural forms rely heavily on:	**Symmetry and tessellation** — Islamic geometric art, crystals, honeycombs, and many natural forms rely on symmetry and tessellation — repeated, symmetric patterns. Generative code can recreate and explore these mathematically-rich structures.
A KALEIDOSCOPE effect in code is made by:	**Reflecting/rotating a pattern into multiple symmetric segments** — A kaleidoscope effect reflects and rotates a base pattern into multiple symmetric segments (wedges) around a center — combining rotational and reflection symmetry to turn a simple pattern into an intricate symmetric design.
BREAKING symmetry intentionally can:	**Create a focal point or dynamic interest** — Intentionally breaking an otherwise symmetric pattern (an off element, a disruption) can create a focal point and dynamic interest — the break stands out against the order. Deliberate asymmetry is a valid design choice.
Symmetry in a design generally creates a feeling of:	**Balance, order, and harmony** — Symmetry creates a feeling of balance, order, and harmony — often calming or formal. Asymmetry (breaking symmetry) creates more dynamic tension. Symmetry is a strong compositional tool for stability.
PREDICT: a shape with 4-fold rotational symmetry looks identical every:	**90° of rotation (360° / 4)** — 4-fold rotational symmetry means the shape matches itself 4 times in a full turn — every 360°/4 = 90°. The order (4) divides 360° to give the rotation angle that maps the shape onto itself.
PREDICT: drawing a shape and its MIRROR across a center line creates:	**A left-right symmetrical (reflected) design** — Drawing a shape and its mirror across a center line creates a left-right symmetrical design — a common, easy way to generate balanced, symmetric compositions in code (for each point at x, draw one at width-x).
ROTATIONAL symmetry means a shape:	**Looks the same after being rotated by some angle** — Rotational symmetry means a shape looks the same after rotation by a certain angle (e.g., a snowflake looks identical every 60°). The order of symmetry is how many times it matches in a full turn.
PREDICT: a grid loop places the SAME tile at every cell. The result is:	**A regular repeating (tessellated) pattern** — A grid loop placing the same tile in every cell produces a regular repeating (tessellated) pattern — the foundation of tiling. Varying the tiles per cell adds interest while keeping the tiling structure.
REFLECTION symmetry means a shape:	**Looks the same on both sides of a mirror line** — Reflection (mirror) symmetry means a shape looks identical on both sides of a mirror line (axis) — one half is the mirror image of the other. A butterfly has reflection symmetry.

14. Making animation feel natural

✂ Question (fold →)	Answer
To make motion FRAME-RATE INDEPENDENT (same speed regardless of fps), you can:	**Scale movement by the time elapsed since the last frame (delta time)** — Scaling movement by delta time (elapsed time per frame) keeps speed consistent regardless of frame rate — so animation runs at the same real-world speed on fast and slow devices. Otherwise, faster fps means faster motion.
Which produces the most NATURAL-feeling arrival at a target?	**Ease-out (decelerating as it approaches)** — Ease-out — decelerating as it approaches — produces the most natural-feeling arrival, settling gently into the target. A constant speed with a sudden stop looks abrupt; accelerating into a target looks unnatural.
LINEAR motion means an object moves:	**At a constant speed (equal steps each frame)** — Linear motion moves an object at constant speed — the same increment each frame (e.g., x += 5). It's steady but can look mechanical/robotic, lacking natural acceleration.
PREDICT: an object accelerating (ease-in) from rest covers ___ distance in early frames vs later frames:	**less distance early, more later** — Ease-in accelerates from rest, so it covers LESS distance in early frames (slow start) and MORE later (as it speeds up). The increasing step size per frame is what makes it 'ease in.'
Motion that loops seamlessly (end matches the start) requires:	**The animation's final state to match its initial state** — A seamless looping animation needs its end state to match its start state, so when it repeats there's no jarring jump — e.g., a full rotation (0° to 360°) or a complete sine cycle loops smoothly.
Easing values are often defined by an EASING FUNCTION that maps:	**Progress (0 to 1) to an eased position (0 to 1)** — An easing function maps linear progress (0 to 1, how far through the animation) to an eased value (0 to 1) — reshaping the timing so motion accelerates/decelerates. Different curves (ease-in, ease-out, bounce) give different feels.
EASING refers to:	**Gradually changing speed (accelerating/decelerating) for natural motion** — Easing gradually changes an object's speed — accelerating (ease-in), decelerating (ease-out), or both — to make motion feel natural and organic rather than mechanically constant (linear).
'Constant (linear) motion always looks the most natural.' This is:	**False — real motion usually accelerates/decelerates; easing looks more natural** — MISCONCEPTION NAMED — 'linear looks natural.' Real-world motion almost always involves acceleration/deceleration; constant linear motion looks mechanical/robotic. Easing (gradual speed change) makes animation feel natural and alive.
FRAME RATE (frames per second, fps) affects animation by:	**Determining how smooth the motion appears** — Frame rate (fps) determines animation smoothness — higher fps (e.g., 60) looks smoother than low fps (choppy). The draw loop runs once per frame, updating and redrawing the scene.
EASE-IN motion:	**Starts slow and accelerates** — Ease-in starts slow and accelerates — a gradual pickup from rest, like an object beginning to move. It's the opposite of ease-out (which decelerates to a stop).
Adding a slight easing or 'bounce' to UI animations makes them feel:	**More polished and satisfying** — Easing (and effects like a subtle bounce/overshoot) make UI and animation feel more polished, natural, and satisfying — mechanical linear motion feels stiff by comparison. Motion design uses easing heavily.

ANIMATION in code is created by:	**Redrawing with slightly changed values each frame** — Animation is created by redrawing the scene each frame with slightly changed values (position, size, color) — rapid successive frames create the illusion of motion, driven by the draw loop.
PREDICT: setting an object's y to sin(time) x 50 makes it:	**Bob up and down smoothly within a 100px range** — y = sin(time) x 50 makes the object bob smoothly up and down between -50 and +50 (a 100px range) as time advances — a sine wave produces natural, smooth oscillating motion.
PREDICT: an object moving a fixed 5 px per frame reaches a target 50 px away in:	**10 frames** — At a constant 5 px/frame over 50 px, the object arrives in 50/5 = 10 frames. Linear motion covers distance predictably (distance = speed x frames).
A common easing technique moves an object a FRACTION of the remaining distance each frame, which produces:	**Ease-out (fast then slowing as the gap shrinks)** — Moving a fraction of the remaining distance each frame (x += (target - x) x 0.1) produces ease-out: fast at first (big gap), slowing as the gap shrinks. A simple, popular easing method.
EASE-OUT motion:	**Starts fast and slows down as it approaches the target** — Ease-out starts fast and decelerates as it nears the target — settling gently into place. It feels natural for objects coming to rest (like a car braking smoothly).
PREDICT: combining horizontal linear motion with vertical sine oscillation produces:	**A wavy path (moving forward while bobbing up and down)** — Combining steady horizontal motion (x increasing linearly) with vertical sine oscillation (y bobbing) produces a wavy path — forward movement with up-and-down waving. Combining motion components creates complex trajectories.
OSCILLATION (back-and-forth motion) is often created using:	**A sine wave function over time** — Oscillation (smooth back-and-forth motion) is commonly created with a sine function of time — sin() smoothly cycles between -1 and 1, producing natural pendulum-like or wave-like movement.
PREDICT: an object with ease-out approaching a target will:	**Slow noticeably as it nears the target, settling smoothly** — An ease-out object slows noticeably as it nears the target, settling in smoothly rather than stopping abruptly — the natural-looking deceleration that makes UI and animation feel polished.
Understanding easing and motion lets a creative coder:	**Create smooth, natural, expressive animation** — Understanding animation (frame-by-frame updates), easing (natural speed changes), and oscillation (sine-based motion) lets a creative coder create smooth, natural, expressive movement — predicting how motion code will look and feel.

15. Transforming existing work into new creations

✂ Question (fold →)	Answer
Variation and remix embody the creative-coding idea that:	**Code is a flexible tool for exploring many possibilities, not one fixed result** — Variation and remix embody the creative-coding mindset: code is a flexible instrument for exploring a space of possibilities, generating many outputs — you design systems and explore them, rather than crafting a single fixed result.
'A variation must change everything about the original.' This is:	**False — a variation can change one aspect while keeping the rest** — MISCONCEPTION NAMED — 'a variation changes everything.' A variation often changes just one aspect (color, tempo, one parameter) while keeping the rest — staying recognizably related to the original. Small, focused changes are the essence of variation.
PREDICT: applying a motif's transformations (transpose, invert) AND changing the instrument creates:	**A remixed variation combining melodic and timbral change** — Combining melodic transformations (transpose, invert) with a change of instrument (timbre) creates a rich remixed variation — layering several kinds of change while the core idea stays recognizable.
Building on others' code (with attribution/permission) is:	**A legitimate, common part of the creative-coding culture** — Building on others' openly-shared code — with proper attribution and within licenses — is a legitimate, common part of creative-coding culture. Remixing and extending shared work is how the community learns and innovates.
Keeping a CONSISTENT STYLE across variations requires:	**Fixing the stylistic parameters while varying others** — To keep a consistent style across variations, fix the stylistic parameters (palette, shape language) while varying others (count, position, seed) — producing a coherent FAMILY of related outputs rather than wildly different ones.
In creative coding, VARIATION means:	**Producing different versions by changing parameters** — Variation produces different versions of a work by changing parameters (color, size, count, seed) — the same code generating many distinct outputs. It's a core strength of parametric, generative code.
Documenting the PARAMETERS/seed of a good variation lets you:	**Reproduce or return to it later** — Documenting the parameters and seed of a good variation lets you reproduce it later (regenerate the exact output) — treating the parameter set as a reproducible 'recipe' for that specific variation.
PREDICT: keeping a generative artwork's code but using a NEW random seed produces:	**A different but stylistically-related output** — Keeping the code but changing the seed produces a new output in the same style — a stylistically-related variation. The code defines the 'style'; the seed selects a specific instance within it.
Which best describes creating a VARIATION of a generative piece?	**Adjusting parameters or the seed to produce a related but different output** — Creating a variation means adjusting parameters or the seed to produce a related-but-different output — reusing the generative system. Rewriting from scratch, deleting, or exact-copying are not variation.
Combining elements from TWO different generative pieces is a form of:	**Remix (recombination)** — Combining elements from two pieces (e.g., one's color scheme with another's shapes) is remix by recombination — creating a new hybrid from existing parts. Recombining is a powerful creative strategy.
Exploring a 'design space'	**Trying many parameter combinations to find good outputs** — Exploring a design space means systematically trying different parameter combinations (and

means:	seeds) to discover good or interesting outputs — the generative artist searches the range of possibilities their code can produce.
PREDICT: reducing a parameter's RANGE (e.g., limiting colors to a small palette) makes the variations:	**More consistent/cohesive with each other** — Reducing a parameter's range (e.g., a smaller palette) makes variations more consistent and cohesive with one another — tighter constraints yield a more unified family of outputs, while wider ranges yield more diverse (or chaotic) results.
Varying MULTIPLE parameters at once creates:	**A wider range of diverse outputs** — Varying multiple parameters at once (color AND count AND size) creates a wider, more diverse range of outputs — the combinations multiply the possibilities in the design space.
PREDICT: a program generates a variation each time a button is pressed (new random values). Pressing it repeatedly gives:	**A series of different variations** — A program generating a new variation (new random values) per button press produces a series of different outputs — letting the user interactively explore the design space, generating variation on demand.
A REMIX takes existing work and:	**Transforms/recombines it into something new** — A remix takes existing work (code, a pattern, a melody) and transforms or recombines it into something new — building on what exists rather than starting from scratch, a central creative-coding practice.
PREDICT: taking a flower-drawing program and changing only the PETAL COUNT parameter produces:	**Flowers with different numbers of petals (variations)** — Changing only the petal-count parameter yields flowers with different petal numbers — clear variations from one program. Isolating a parameter to vary is how generative artists explore a design space.
In music, a REMIX of a melody might:	**Change its tempo, instrument, or rhythm while keeping the core tune** — A music remix might change tempo, instrumentation, or rhythm while keeping the recognizable core melody — transforming the feel while preserving the identity. It's variation applied to reimagine a piece.
Understanding remix and variation lets a creative coder:	**Explore a design space and generate families of related creative works** — Understanding remix and variation lets a creative coder explore a design space — varying parameters, seeds, and combinations — to generate rich families of related works, embracing code as a system for creative exploration.
A 'generative system' is valuable because it can:	**Produce endless unique variations from one set of rules** — A generative system's value is producing endless unique variations from one set of rules/code — each output distinct yet related. The artist designs the SYSTEM (rules + parameters), and it generates the many artworks.
Because generative code is PARAMETRIC, creating a variation is often as simple as:	**Changing an input value (a parameter or seed)** — Parametric generative code makes variation easy — change a parameter (color, count) or the random seed, and you get a new output instantly, without rewriting the underlying logic. This is a superpower of generative art.

16. Integrating every creative-coding concept

✂ Question (fold →)	Answer
Which best integrates art and music concepts?	**A seeded loop that plays a scale-based melody AND draws each note mapped to color and position** — A seeded loop playing a scale-based melody while drawing each note mapped to color and position integrates nearly every concept — iteration, pitch/scale, randomness/seed, color, mapping, and sync. The others use only one domain or lack meaningful connection.
PREDICT: mapping PITCH to vertical position (higher notes -> higher on screen) creates:	**A visual representation of the melody's contour** — Mapping pitch to vertical position (higher notes higher up) visualizes the melody's contour — the rising and falling shape of the tune becomes visible. Data/parameter mapping links the musical and visual domains.
Randomness (with a seed) in an audiovisual piece lets you:	**Generate reproducible variations of both the art and music** — Using seeded randomness across an audiovisual piece generates reproducible variations of both the visuals and the music together — the same seed regenerates the entire synchronized piece, art and sound alike.
Using ONE loop to drive both a visual pattern and a rhythmic pattern:	**Keeps the art and music synchronized** — Driving both visuals and rhythm from one loop/clock keeps them synchronized — each beat updates both the sound and the image together, so the art visibly moves with the music. Shared timing is key to audiovisual sync.
Every creative-coding concept (loops, functions, variables, randomness, transforms, mapping) serves the goal of:	**Generating expressive art and music from code** — Every concept in this bank — iteration, functions/parameters, variables/state, randomness, color, transforms, rhythm, pitch, mapping — is a tool serving the ultimate goal: generating expressive, original art and music from code.
Structuring an audiovisual piece with SECTIONS (like verse/chorus) requires:	**Changing parameters/patterns over time to mark each section** — Structuring an audiovisual piece into sections requires changing parameters/patterns over time (e.g., new palette + new rhythm for a 'chorus') — contrast marks the sections, giving the piece form and development.
PREDICT: increasing the TEMPO of the music in a synced piece will make the visuals:	**Update/animate faster (in sync with the faster beat)** — If visuals are synced to the beat, increasing the tempo makes both the music and the visuals update faster together — the shared timing means a tempo change drives the whole synchronized piece faster.
PREDICT: a generative audiovisual system with a seed + adjustable parameters can produce:	**Countless reproducible, synchronized art-and-music variations** — A generative audiovisual SYSTEM (seed + adjustable parameters driving both domains) can produce countless reproducible, synchronized variations of art and music — designing the system, then exploring its space, is the capstone of creative coding.
A loop that iterates over a MELODY's notes can simultaneously:	**Play each note AND draw a corresponding visual element** — A loop over a melody's notes can, per note, both play the note and draw a visual element (position/color mapped from pitch/timing) — generating synchronized audio and visuals from one iteration. Integration through a shared loop.
PREDICT: a piece where visuals are generated with NO connection to the music will feel:	**Disjointed (the art and music don't relate)** — Visuals generated with no connection to the music feel disjointed — the two seem unrelated. Meaningful mapping (linking musical events to visual changes) is what makes an audiovisual piece feel unified and synchronized.

PREDICT: an interactive audiovisual piece where mouse movement changes BOTH the visuals and the music lets the user:	**Perform/shape the whole audiovisual experience in real time** — An interactive audiovisual piece mapping mouse movement to both visuals and music lets the user perform/shape the entire experience live — combining interaction, mapping, and generation across both domains.
An audiovisual generative piece combines:	**Generated visuals AND generated/synced sound** — An audiovisual generative piece combines generated visuals with generated or synchronized sound — integrating the visual concepts (shape, color, motion) with the musical ones (rhythm, pitch, harmony) into one work.
The ultimate goal of creative coding is to:	**Use code as an expressive medium to generate original art and music** — The ultimate goal of creative coding is to use code as an expressive medium — combining iteration, functions, variables, randomness, transforms, color, rhythm, pitch, mapping, and interaction to generate original, expressive art and music. Every concept in this bank serves that creative end.
PREDICT: mapping the music's VOLUME to visual brightness makes the visuals:	**Brighter during loud passages, dimmer during quiet ones** — Mapping volume to brightness makes visuals brighter in loud passages and dimmer in quiet ones — the visuals react to the music's dynamics. Parameter mapping ties visual properties to musical properties.
To make visuals feel CONNECTED to a melody, you might map:	**The melody's pitch, rhythm, and dynamics to visual position, timing, and brightness** — Mapping multiple musical parameters (pitch -> position, rhythm -> timing/motion, dynamics -> brightness) to visual properties makes the visuals feel deeply connected to the melody — richer mapping produces a more unified, expressive audiovisual piece.
The core skill of audiovisual coding is:	**Mapping between musical and visual parameters meaningfully** — The core skill of audiovisual coding is mapping between musical and visual parameters meaningfully — deciding what sound property drives what visual property (pitch->height, beat->pulse, volume->brightness) to create a coherent, synchronized experience.
'Art and music coding use completely unrelated concepts.' This is:	**False — both use iteration, parameters, variables, randomness, and structure** — MISCONCEPTION NAMED — 'art and music coding are unrelated.' Both rely on the same core concepts — iteration (loops), functions/parameters, variables/state, randomness, and structure. This shared foundation is exactly why they combine so naturally.
PREDICT: syncing a shape's size to a drum beat makes the shape:	**Pulse/grow on each beat** — Syncing a shape's size to a drum beat makes it pulse/grow on each beat — the audio event drives a visual parameter. Mapping music to visuals is the heart of audiovisual generative work.
Combining a GRID of visuals with a step-sequencer rhythm can create:	**A visual step sequencer where each cell corresponds to a beat/sound** — Combining a visual grid with a step-sequencer rhythm can make each grid cell correspond to a beat/sound — lighting up cells in time with the music. Shared structure (the grid = the sequencer steps) unifies art and music.
Using FUNCTIONS to organize an audiovisual program helps by:	**Keeping the visual and audio logic modular and reusable** — Functions organize an audiovisual program into modular, reusable pieces (drawScene(), playBeat(), etc.) — keeping the visual and audio logic manageable and letting you compose complex behavior from clear building blocks.

Keep going

You practiced **320 cards** from GenForge. Come back to the ones you missed — spaced-out practice makes them stick.

Spark & Anvil is a 501(c)(3) public charity. Every app, story, and book is free, forever — no ads, no in-app purchases, no tracking.

Keep exploring online

Play it now	More free books
`https://spark-and-anvil.org/play/genforge`	`https://spark-and-anvil.org/books`

Scan a code with any phone camera, or type the address. Every app, story, and book is free, forever — no account, no tracking.