

UNIT 3

Creating Types in C#

(12 Hours) — C# Programming

Unit I introduced the runtime that executes C# programs. This unit turns to the language itself, and specifically to the mechanisms C# gives you for defining your own types: classes, structs, interfaces, and enums. Every non-trivial C# program is, at its core, a collection of custom types collaborating with one another, so mastering how to declare, construct, and compose them is the single most important skill in the language.

1. Classes

A class is the most fundamental building block for custom types in C#. It is a template that combines fields (data) and members (functions, properties, events) that operate on that data into a single reference type. Every object you create from a class is called an instance, and each instance has its own independent copy of the class's instance fields.

```
public class Account
{
    // Fields (data)
    private decimal balance;
    // Methods (behaviour)
    public void Deposit(decimal amount)
    {
        balance += amount;
    }
}
```

Classes are declared with the `class` keyword and are reference types: a variable of a class type holds a reference (pointer) to an object on the heap, not the object's data directly. Assigning one class variable to another copies the reference, so both variables end up pointing at the same underlying object.

Example — Reference Semantics

```
Account a = new Account();
a.Deposit(100);
Account b = a; // b now refers to the SAME object as a
b.Deposit(50);
// a.Balance is now 150 too, because a and b share one instance
```

Note: Contrast this with struct (Section 15), which is a value type: assigning a struct variable copies the entire value, producing two independent copies.

2. Constructors and Deconstructors

2.1 Constructors

A constructor is a special method that runs automatically when an instance of a class is created with `new`. It has the same name as the class, no return type, and is typically used to initialize fields to valid starting values.

```
public class Account
{
    private decimal balance;
    private string owner;
    public Account(string owner, decimal openingBalance)
    {
        this.owner = owner;
        balance = openingBalance;
    }
}
Account acc = new Account("Rita", 500m); // constructor runs here
```

A class may define multiple constructors with different parameter lists — this is simply an application of method overloading (Section 13). One constructor can call another on the same class using the `this(...)` constructor initializer, avoiding duplicated initialization logic.

```
public class Account
{
    private decimal balance;
    public Account() : this("Unnamed", 0m) { }
    public Account(string owner, decimal balance)
    {
        this.owner = owner;
        this.balance = balance;
    }
}
```

2.2 Deconstructors

A deconstructor (not to be confused with a destructor/finalizer, Section 7) lets an object be unpacked into a set of separate variables in a single expression, mirroring how a constructor packs values into an object. It is declared as a `Deconstruct` method with one or more `out` parameters.

```

public class Point
{
    public double X { get; }
    public double Y { get; }
    public Point(double x, double y) { X = x; Y = y; }
    public void Deconstruct(out double x, out double y)
    {
        x = X;
        y = Y;
    }
}
Point p = new Point(3, 4);
var (px, py) = p; // calls Deconstruct automatically
Console.WriteLine($"{px}, {py}"); // 3, 4

```

Note: Deconstruction works especially well with tuples and pattern matching, and a class can define several overloads of Deconstruct with different numbers of output parameters.

3. The this Reference

Inside any instance member, the keyword `this` refers to the specific instance on which that member is currently executing. It has two principal uses in C#.

- Disambiguation — when a parameter or local variable has the same name as a field, `this.fieldName` explicitly refers to the field.
- Self-reference — passing the current instance to another method, or returning it to support method chaining (a fluent interface).

```

public class Builder
{
    private string text = "";
    public Builder Append(string s)
    {
        this.text += s; // 'this.' clarifies we mean the field
        return this;   // returning 'this' enables chaining
    }
}
var result = new Builder().Append("Hello, ").Append("world!");

```

Note: this cannot be used inside a static member, because static members are not associated with any particular instance (see Section 6).

4. Properties

A property looks like a public field from the outside but is implemented with accessor methods — a `get` accessor to read the value and a `set` accessor to write it. Properties let a class expose data while retaining full control over validation, computation, or side effects, which is central to encapsulation.

```
public class Account
{
    private decimal balance;
    public decimal Balance
    {
        get { return balance; }
        set
        {
            if (value < 0)
                throw new ArgumentException("Balance cannot be negative.");
            balance = value;
        }
    }
}
```

4.1 Auto-Implemented Properties

When a property needs no custom logic, C# lets you omit the backing field entirely and have the compiler generate it automatically.

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

4.2 Read-Only and Init-Only Properties

Omitting the `set` accessor makes a property read-only from outside the class — its value can only be assigned inside a constructor (or where declared). The `init` accessor (C# 9+) permits assignment only during object initialization, supporting immutable objects while still allowing object-initializer syntax.

```
public class Person
{
    public string Name { get; init; }    // settable only at construction time
    public int Age { get; }              // fully read-only
    public Person(int age) { Age = age; }
}
var p = new Person(30) { Name = "Kiran" }; // OK – object initializer
// p.Name = "Other";                      // compile error after this
```

Key Takeaways — Sections 1–4

- ✓ Classes are reference types combining data and behaviour into instances.
- ✓ Constructors initialize new instances; deconstructors unpack them via out parameters.
- ✓ `this` refers to the current instance, disambiguating names and enabling chaining.
- ✓ Properties wrap field access in get/set logic while preserving field-like syntax.
- ✓ Auto-properties and init accessors reduce boilerplate for simple or immutable types.

5. Indexers

An indexer allows an object to be indexed using array-style square bracket syntax, `obj[index]`. It is declared using the `this` keyword in place of a method name, together with get and set accessors similar to a property.

```
public class Sentence
{
    private string[] words;
    public Sentence(string text) => words = text.Split(' ');
    public string this[int index]
    {
        get => words[index];
        set => words[index] = value;
    }
    public int Length => words.Length;
}
var s = new Sentence("the quick fox");
Console.WriteLine(s[1]); // quick
s[1] = "slow";           // uses the set accessor
```

Indexers can be overloaded to accept different key types (an int, a string, a custom struct), and they may take multiple parameters, mimicking a multi-dimensional lookup.

Note: Indexers are what allow familiar bracket syntax on types such as `List<T>` and `Dictionary<TKey, TValue>` from the Base Class Library.

6. Static Constructors and Classes

6.1 Static Members

A member marked `static` belongs to the type itself rather than to any individual instance. There is exactly one copy of a static field, shared by the entire application, regardless of how many instances of the class exist (or even if none exist at all).

```
public class Counter
{
    public static int InstancesCreated = 0;
    public Counter() => InstancesCreated++;
}
new Counter(); new Counter(); new Counter();
Console.WriteLine(Counter.InstancesCreated);    // 3
```

6.2 Static Constructors

A static constructor initializes static data for a class. It takes no parameters, cannot be called explicitly, and the runtime guarantees it executes automatically, at most once, before the class is first used — either when the first instance is created or the first static member is accessed, whichever happens first.

```
public class Configuration
{
    public static readonly string ConnectionString;
    static Configuration()
    {
        ConnectionString = LoadFromFile();    // runs once, automatically
    }
    private static string LoadFromFile() => "Server=...";
}
```

6.3 Static Classes

A class declared `static` can contain only static members, cannot be instantiated with `new`, and cannot be used as a base class. Static classes are the natural home for pure utility/helper functionality — the built-in `System.Math` and `System.Console` classes are both static.

```
public static class MathHelpers
{
    public static int Square(int x) => x * x;
}
int result = MathHelpers.Square(5);    // called on the type, no instance needed
```

7. Finalizers

A finalizer (historically called a destructor, using `~ClassName()` syntax) is a special method the garbage collector calls before reclaiming an object's memory, giving it a last chance to release unmanaged resources such as file handles or native memory that the GC does not know how to clean up on its own.

```
public class FileWrapper
{
    private IntPtr handle;    // an unmanaged resource
    ~FileWrapper()           // finalizer
    {
        // Release 'handle' here as a last resort
        ReleaseHandle(handle);
    }
}
```

- Finalizers run at a time chosen by the garbage collector, not deterministically — you cannot know exactly when (or guarantee that) a finalizer executes before the program ends.
- Objects with finalizers require at least two garbage-collection passes to be fully reclaimed, which is measurably slower than ordinary objects.
- The recommended pattern is to implement `IDisposable` for deterministic cleanup, and use a finalizer only as a safety net in case `Dispose()` was never called.

Note: Because of their cost and unpredictability, modern C# guidance is to avoid writing your own finalizer unless a class directly owns an unmanaged handle; prefer the `Dispose` pattern together with the `using` statement wherever possible.

8. Dynamic Binding

C# resolves most method calls at compile time — this is called static binding. The `dynamic` keyword (introduced in C# 4.0) instead defers member resolution to run time, which is dynamic binding.

```
dynamic value = GetValueFromSomewhere();  
value.DoSomething(); // resolved at run time, not compile time
```

With a `dynamic` variable, the compiler performs no type checking at all on member access — it simply trusts that the operation will succeed and defers the actual lookup to the Dynamic Language Runtime (DLR) when the line actually executes. If the member does not exist, the error surfaces as a run-time exception (`RuntimeBinderException`) rather than a compile-time error.

Aspect	Static Binding (normal types)	Dynamic Binding (dynamic)
When resolved	Compile time	Run time
Errors caught	At compile time	At run time, as exceptions
IntelliSense support	Full	None
Typical use case	Ordinary application code	COM interop, reflection-heavy or dynamic-language scenarios

Note: Dynamic binding trades compile-time safety for flexibility. It is invaluable for interoperating with dynamically typed systems (like scripting engines or older COM APIs) but should not replace normal static typing in everyday code.

Key Takeaways — Sections 5–8

- ✓ Indexers give objects array-style `this[]` access using `get/set` accessors.
- ✓ static members belong to the type; static constructors run once, automatically, before first use.
- ✓ static classes hold only static members and cannot be instantiated.
- ✓ Finalizers give the GC a last chance to clean up unmanaged resources, but run non-deterministically.
- ✓ `dynamic` defers member resolution to run time, sacrificing compile-time checking for flexibility.

9. Operator Overloading

C# allows a class or struct to redefine the behaviour of built-in operators (such as `+`, `-`, `==`) when applied to instances of that type, using the `operator` keyword. This lets custom types support natural, mathematical-looking syntax.

```

public struct Vector2
{
    public double X, Y;
    public Vector2(double x, double y) { X = x; Y = y; }
    public static Vector2 operator +(Vector2 a, Vector2 b)
        => new Vector2(a.X + b.X, a.Y + b.Y);
    public static bool operator ==(Vector2 a, Vector2 b)
        => a.X == b.X && a.Y == b.Y;
    public static bool operator !=(Vector2 a, Vector2 b) => !(a == b);
    public override bool Equals(object obj) => obj is Vector2 v && this == v
;
    public override int GetHashCode() => (X, Y).GetHashCode();
}
var sum = new Vector2(1, 2) + new Vector2(3, 4);    // (4, 6)

```

- Operators must be declared public and static.
- Comparison operators are conventionally overloaded in pairs: == with !=, and < with >.
- When == is overloaded, Equals and GetHashCode should also be overridden consistently to avoid surprising behaviour in collections such as Dictionary<TKey,TValue>.

Note: Operator overloading should be used sparingly and only when the operator's meaning is intuitive for the type — overloading + for a Vector2 makes sense; overloading it to mean something unrelated to addition would confuse readers.

10. Inheritance

Inheritance lets one class (the derived or subclass) acquire the members of another class (the base or superclass), expressing an "is-a" relationship. C# supports only single inheritance of classes — a class may derive from exactly one base class — though it may implement any number of interfaces (Section 17).

```

public class Animal
{
    public string Name { get; set; }
    public void Eat() => Console.WriteLine($"{Name} is eating.");
}
public class Dog : Animal    // Dog inherits from Animal
{
    public void Bark() => Console.WriteLine($"{Name} says Woof!");
}
var d = new Dog { Name = "Rex" };
d.Eat();    // inherited from Animal
d.Bark();   // defined in Dog

```

A derived class inherits every accessible member of its base class and can additionally define new members, override virtual members (see below), or hide base members entirely. Every class in C# ultimately derives, directly or indirectly, from `System.Object` — reinforcing the unified type system introduced in Unit I.

10.1 virtual and override

By default, methods are not polymorphic: calling a method through a base class reference always executes the base class's version. Marking a base method `virtual` and a derived method `override` allows the derived implementation to run even when accessed through a base-typed reference — the essence of runtime polymorphism.

```
public class Animal
{
    public virtual void Speak() => Console.WriteLine("...");
}
public class Dog : Animal
{
    public override void Speak() => Console.WriteLine("Woof!");
}
Animal a = new Dog();
a.Speak();    // "Woof!" — the DERIVED method runs, thanks to virtual/override
```

Example — Why virtual Matters

Without `virtual/override`, `Animal a = new Dog(); a.Speak();` would print `"..."` because the compiler would bind the call to `Animal.Speak()` at compile time based on the declared type of `'a'`, ignoring the fact that the actual object is a `Dog`. `virtual/override` defers that decision to run time, based on the object's real type — this is dynamic dispatch.

11. Abstract Classes and Methods

An abstract class is a class that cannot be instantiated directly — it exists purely to be a base for other classes. It may contain a mix of fully implemented members and abstract members: method signatures with no body at all, which every non-abstract derived class is required to implement.

```

public abstract class Shape
{
    public abstract double Area();           // no body — must be overridden
    public void Describe()                  // ordinary, inherited as-is
        => Console.WriteLine($"Area = {Area()}");
}
public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area() => Math.PI * Radius * Radius;
}
// Shape s = new Shape();    // COMPILE ERROR — cannot instantiate abstract c
// lass
Shape c = new Circle { Radius = 2 };
c.Describe();    // Area = 12.566...

```

- Abstract methods are implicitly virtual and must be overridden in any concrete (non-abstract) derived class.
- An abstract class can still have constructors, fields, and fully implemented methods — it simply cannot be instantiated on its own.
- Abstract classes are appropriate when derived types share substantial common implementation, not just a common contract (compare with interfaces, Section 17).

Note: Trying to instantiate an abstract class directly, or forgetting to override an abstract member in a concrete subclass, is a compile-time error — the compiler enforces the contract for you.

12. The base Keyword

The `base` keyword lets a derived class refer explicitly to members of its immediate base class, most commonly to call a base class constructor or to invoke the base implementation of an overridden method from within the override itself.

```

public class Animal
{
    public string Name { get; }
    public Animal(string name) { Name = name; }
    public virtual void Speak() => Console.WriteLine("...");
}
public class Dog : Animal
{
    public Dog(string name) : base(name) { } // calls Animal's constructor
    public override void Speak()
    {
        base.Speak(); // run the base version first
        Console.WriteLine("Woof!");
    }
}

```

Note: Unlike this(...), which chains to another constructor on the SAME class, base(...) always targets a constructor on the immediate BASE class, and it is the mechanism by which base-class initialization is guaranteed to run before any derived-class constructor body executes.

Key Takeaways — Sections 9–12

- ✓ Operator overloading lets custom types support natural operator syntax via public static operator methods.
- ✓ Inheritance expresses is-a relationships; C# supports single inheritance of classes only.
- ✓ virtual + override enable runtime polymorphism through dynamic dispatch.
- ✓ Abstract classes define partial contracts that concrete subclasses must complete.
- ✓ base accesses the immediate base class's constructor or overridden members explicitly.

13. Overloading

Method overloading allows a type to declare multiple methods that share the same name but differ in their parameter list (number, order, or types of parameters). The compiler chooses the correct overload to call based on the arguments supplied at each call site — this decision is made at compile time and is a form of compile-time (static) polymorphism, distinct from the runtime polymorphism seen with virtual/override.

```

public class Printer
{
    public void Print(int i) => Console.WriteLine($"int: {i}");
    public void Print(string s) => Console.WriteLine($"string: {s}");
    public void Print(double d, int decimals) =>
        Console.WriteLine(d.ToString("F" + decimals));
}

var p = new Printer();
p.Print(5);           // calls Print(int)
p.Print("hello");     // calls Print(string)
p.Print(3.14159, 2);  // calls Print(double, int)

```

- Overloads cannot differ by return type alone — the parameter lists must differ.
- Constructors are overloaded the same way ordinary methods are (Section 2).
- Operator overloading (Section 9) is a specialised application of the same underlying idea, applied to operator symbols instead of method names.

14. The Object Type

`System.Object` (aliased as `object` in C#) is the ultimate base type of every type in the CLR's type system — every class, struct, enum, delegate, and array implicitly derives from it, directly or indirectly. This is the "unified type system" referenced in Unit I.

Member	Purpose
<code>ToString()</code>	Returns a human-readable string representation of the object
<code>Equals(object)</code>	Determines whether two objects are considered equal
<code>GetHashCode()</code>	Returns an integer hash used by hash-based collections
<code>GetType()</code>	Returns runtime type information (a <code>System.Type</code> instance)

Because value types (`int`, `double`, structs) also derive from `Object`, a value type can always be converted to `object` — a process called **boxing** — and converted back with **unboxing**.

```

int number = 42;
object boxed = number;           // boxing: value copied onto the heap
int unboxed = (int)boxed;        // unboxing: value copied back
Console.WriteLine(number.GetType()); // System.Int32

```

Note: Boxing and unboxing involve copying and heap allocation, so they carry a measurable performance cost; generics (Section 19) were introduced largely to let code avoid unnecessary boxing when working with value types.

15. Structs

A struct is declared much like a class but is a value type rather than a reference type. Assigning, passing, or returning a struct copies its entire contents, rather than copying a reference to shared storage.

```
public struct Point
{
    public int X, Y;
    public Point(int x, int y) { X = x; Y = y; }
}
Point p1 = new Point(1, 2);
Point p2 = p1;          // p2 is a full, independent COPY of p1
p2.X = 99;
Console.WriteLine(p1.X); // still 1 — p1 was unaffected
```

Aspect	class (reference type)	struct (value type)
Storage	Heap (referenced from the stack)	Typically stack, or inline within containing type
Assignment	Copies the reference	Copies the full value
Default value	null	All fields set to their own defaults
Inheritance	Supports single inheritance	Cannot inherit from another struct or class
Typical use	Larger, mutable, identity-based entities	Small, immutable, value-based data (Point, Money)

Structs are most appropriate for small, immutable clusters of data where value semantics (copy-on-assignment) and avoiding heap allocation are beneficial — the built-in numeric types themselves, along with types like `DateTime` and `decimal`, are all structs.

16. Access Modifiers

Access modifiers control which parts of a program are permitted to see and use a given type or member, enforcing encapsulation at the language level.

Modifier	Accessible From
public	Anywhere — no restriction
private	Only within the same class (or struct)
protected	The declaring class and any class derived from it
internal	Anywhere within the same assembly (project)

Modifier	Accessible From
protected internal	Same assembly, OR any derived class in any assembly
private protected	Derived classes, but only within the same assembly

```
public class BankAccount
{
    private decimal balance;           // hidden from everything outside
    protected void LogActivity() { }   // usable by BankAccount and subclasses
    public decimal GetBalance() => balance; // usable by anyone
}
```

Note: If no access modifier is specified, class members default to private and top-level types default to internal — a deliberate design choice that encourages the most restrictive (safest) visibility unless you opt into something broader.

Key Takeaways — Sections 13–16

- ✓ Overloading resolves which method to call at compile time, based on parameter lists.
- ✓ System.Object is the root of every type; ToString, Equals, GetHashCode, GetType are universal.
- ✓ Boxing converts a value type to object; unboxing converts it back, with a performance cost.
- ✓ Structs are value types with copy semantics, ideal for small, immutable data.
- ✓ Access modifiers (public, private, protected, internal, etc.) enforce encapsulation boundaries.

17. Interfaces

An interface defines a pure contract: a set of members (methods, properties, events, indexers) that an implementing type promises to provide, without dictating how those members are implemented. Unlike classes, a type may implement any number of interfaces, which is how C# achieves the effect of multiple inheritance for behaviour contracts.

```

public interface IShape
{
    double Area();
    double Perimeter();
}
public class Rectangle : IShape
{
    public double Width, Height;
    public double Area() => Width * Height;
    public double Perimeter() => 2 * (Width + Height);
}
IShape shape = new Rectangle { Width = 3, Height = 4 };
Console.WriteLine(shape.Area()); // 12

```

Aspect	Interface	Abstract Class
Implementation	Traditionally none (default methods allowed since C# 8)	Can mix implemented and abstract members
Multiple inheritance	A type may implement many interfaces	A class may derive from only one base class
Fields	Cannot declare instance fields	Can declare fields freely
Use case	Defining a capability/contract (IComparable, IDisposable)	Sharing substantial common implementation

Many core BCL abstractions are interfaces — `IEnumerable<T>`, `IComparable<T>`, and `IDisposable` are prime examples — precisely because they describe a capability that many unrelated types can share without needing a common base class.

Note: Since C# 8.0, interfaces may include default method implementations, blurring the line with abstract classes somewhat — but interfaces still cannot hold instance state (fields), which remains the key distinction.

18. Enums

An enum (enumeration) defines a closed set of named integral constants, giving a meaningful name to what would otherwise be a collection of "magic numbers" scattered through code.

```

public enum DayOfWeek
{
    Sunday,      // 0
    Monday,      // 1
    Tuesday,     // 2
    Wednesday,   // 3
    Thursday,    // 4
    Friday,      // 5
    Saturday     // 6
}
DayOfWeek today = DayOfWeek.Wednesday;
if (today == DayOfWeek.Saturday || today == DayOfWeek.Sunday)
    Console.WriteLine("Weekend!");

```

By default, the underlying type of an enum is int and values start at 0, incrementing by 1, but both can be customized explicitly.

```

[Flags]
public enum FilePermissions : byte
{
    None      = 0,
    Read      = 1,    // 0001
    Write     = 2,    // 0010
    Execute   = 4,    // 0100
}
var perms = FilePermissions.Read | FilePermissions.Write;    // combine flags
bool canRead = (perms & FilePermissions.Read) != 0;

```

Note: The [Flags] attribute signals that an enum's values are meant to be combined bitwise, and its members should therefore be powers of two so each flag occupies a distinct bit.

19. Generics

Generics allow a class, interface, or method to be written once, parameterized over one or more types, and then reused with any concrete type supplied at the call site — without sacrificing compile-time type safety or resorting to boxing/casting through object.

```
public class Box<T>
{
    private T item;
    public void Store(T value) => item = value;
    public T Retrieve() => item;
}
var intBox = new Box<int>();
intBox.Store(42);
int x = intBox.Retrieve();           // no cast needed, fully type-checked
var stringBox = new Box<string>();
stringBox.Store("hello");
```

19.1 Why Generics Matter

Before generics, a general-purpose collection like `ArrayList` stored everything as object, forcing callers to cast on every retrieval and incurring boxing costs for value types. Generic collections such as `List<T>` and `Dictionary<TKey, TValue>` solve both problems at once.

```
List<int> numbers = new List<int>();
numbers.Add(10);
int first = numbers[0];    // strongly typed – no cast, no boxing
```

19.2 Generic Constraints

A `where` clause restricts which types may be substituted for a type parameter, enabling the compiler to allow operations that would not be safe for every possible type.

```
public class Repository<T> where T : class, new()
{
    public T CreateNew() => new T();    // legal only because of 'new()' constraint
}
public static T Max<T>(T a, T b) where T : IComparable<T>
    => a.CompareTo(b) > 0 ? a : b;    // legal because of IComparable<T> constraint
```

Constraint	Meaning
where T : class	T must be a reference type
where T : struct	T must be a non-nullable value type
where T : new()	T must have an accessible parameterless constructor
where T : BaseType	T must be BaseType or derive from it

Constraint	Meaning
where T : Interface	T must implement the given interface

Generics can also be applied to methods independently of the enclosing type, and to interfaces and delegates. The result is code that is both reusable across many types and fully verified by the compiler at the point of use — the culmination of the type-safety principles introduced at the very start of this course.

Key Takeaways — Sections 17–19

- ✓ Interfaces define pure contracts; a type can implement any number of them.
- ✓ Enums give meaningful names to sets of related integral constants.
- ✓ The [Flags] attribute supports bitwise-combinable enum values.
- ✓ Generics parameterize types and methods over a type argument, avoiding casts and boxing.
- ✓ Constraints (where T : ...) let generic code safely use operations specific to certain types.

Overall Unit 3 Summary

- ✓ Classes bundle data and behaviour into reference-type instances; structs do the same as value types.
- ✓ Constructors, this, properties, and indexers control how instances are created and accessed.
- ✓ static members/constructors/classes belong to the type itself, not to any one instance.
- ✓ Inheritance, virtual/override, abstract classes, and base together enable runtime polymorphism.
- ✓ Overloading gives compile-time polymorphism; operator overloading extends it to symbols.
- ✓ Access modifiers enforce encapsulation; interfaces and enums add contract- and constant-based design.
- ✓ Generics tie the unit together, letting types and methods be reused safely across many concrete types.

— End of Unit 3 —