Programming in C#

Inheritance and Polymorphism

C# Classes

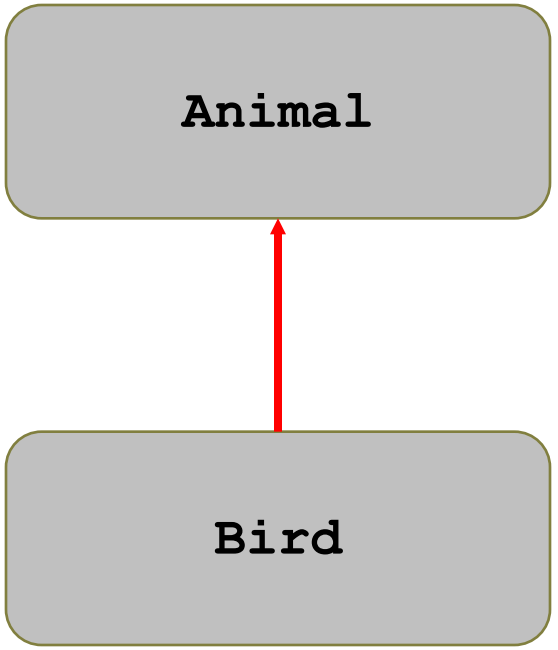
- Classes are used to accomplish:
  - Modularity: Scope for global (static) methods
  - Blueprints for generating objects or instances:
    - Per instance data and method signatures
- Classes support
  - Data encapsulation - private data and implementation.
  - Inheritance - code reuse

Inheritance

- Inheritance allows a software developer to derive a new class from an existing one.
- The existing class is called the parent, super, or base class.
- The derived class is called a child or subclass.
- The child inherits characteristics of the parent.
  - Methods and data defined for the parent class.
- The child has special rights to the parents methods and data.
  - Public access like any one else
  - *Protected* access available only to child classes (and their descendants).
- The child has its own unique behaviors and data.

Inheritance

- Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class.
- Inheritance should create an ***is-a*** relationship, meaning the child *is a* more specific version of the parent.



Examples: Base Classes and Derived Classes

Base class	Derived classes
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

Declaring a Derived Class

- Define a new class DerivedClass which extends BaseClass

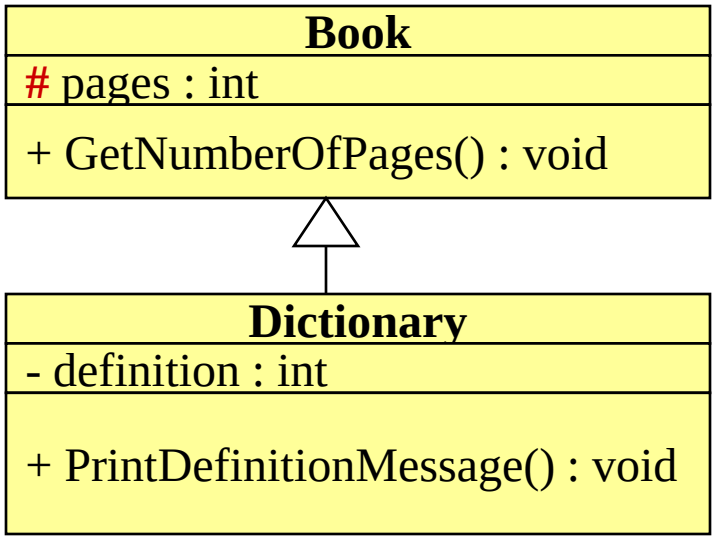
```
class BaseClass
{
    // class contents
}
class DerivedClass : BaseClass
{
    // class contents
}
```

Controlling Inheritance

- A child class inherits the methods and data defined for the parent class; however, whether a data or method member of a parent class is accessible in the child class depends on the visibility modifier of a member.
- Variables and methods declared with *private* visibility are not accessible in the child class
  - However, a private data member defined in the parent class is still part of the state of a derived class.
- Variables and methods declared with *public* visibility are accessible; but public variables violate our goal of encapsulation
- There is a third visibility modifier that helps in inheritance situations: *protected.*

The protected Modifier

- Variables and methods declared with protected visibility in a parent class are only accessible by a child class or any class derived from that class



+ public
- private
protected

Single Inheritance

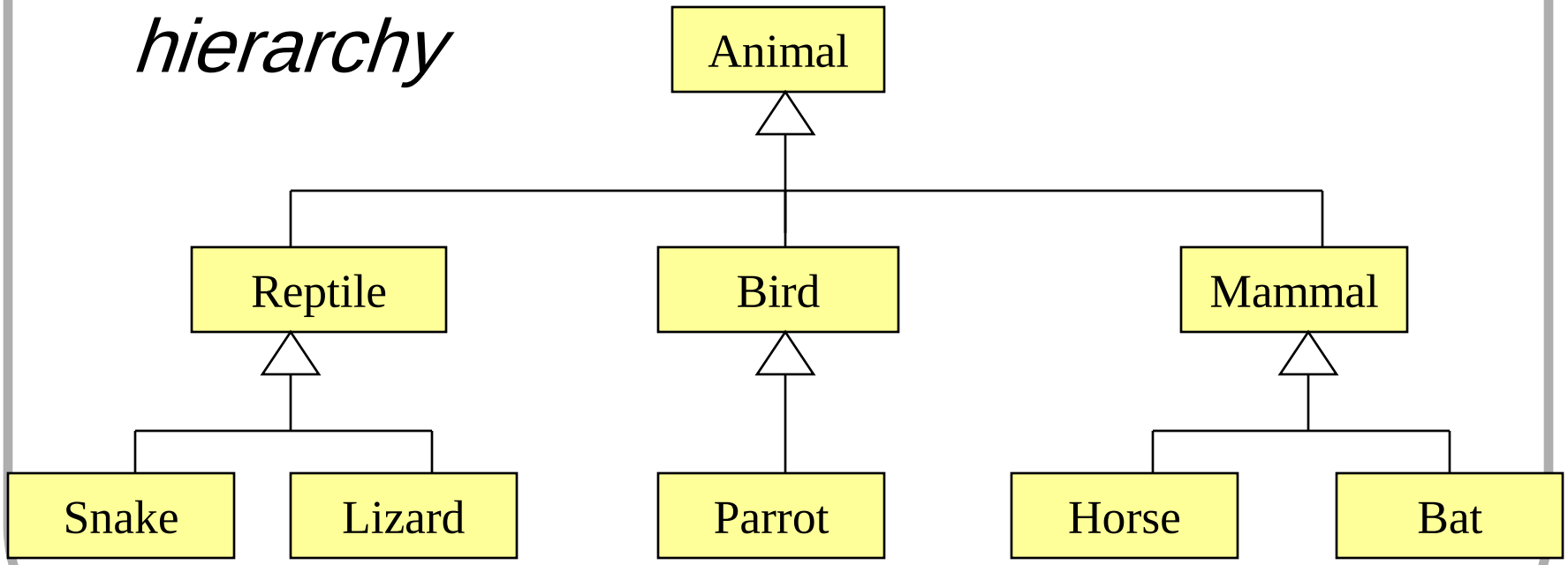
- Some languages, e.g., C++, allow *Multiple inheritance*, which allows a class to be derived from two or more classes, inheriting the members of all parents.
- C# and Java support *single inheritance*, meaning that a derived class can have only one parent class.

Overriding Methods

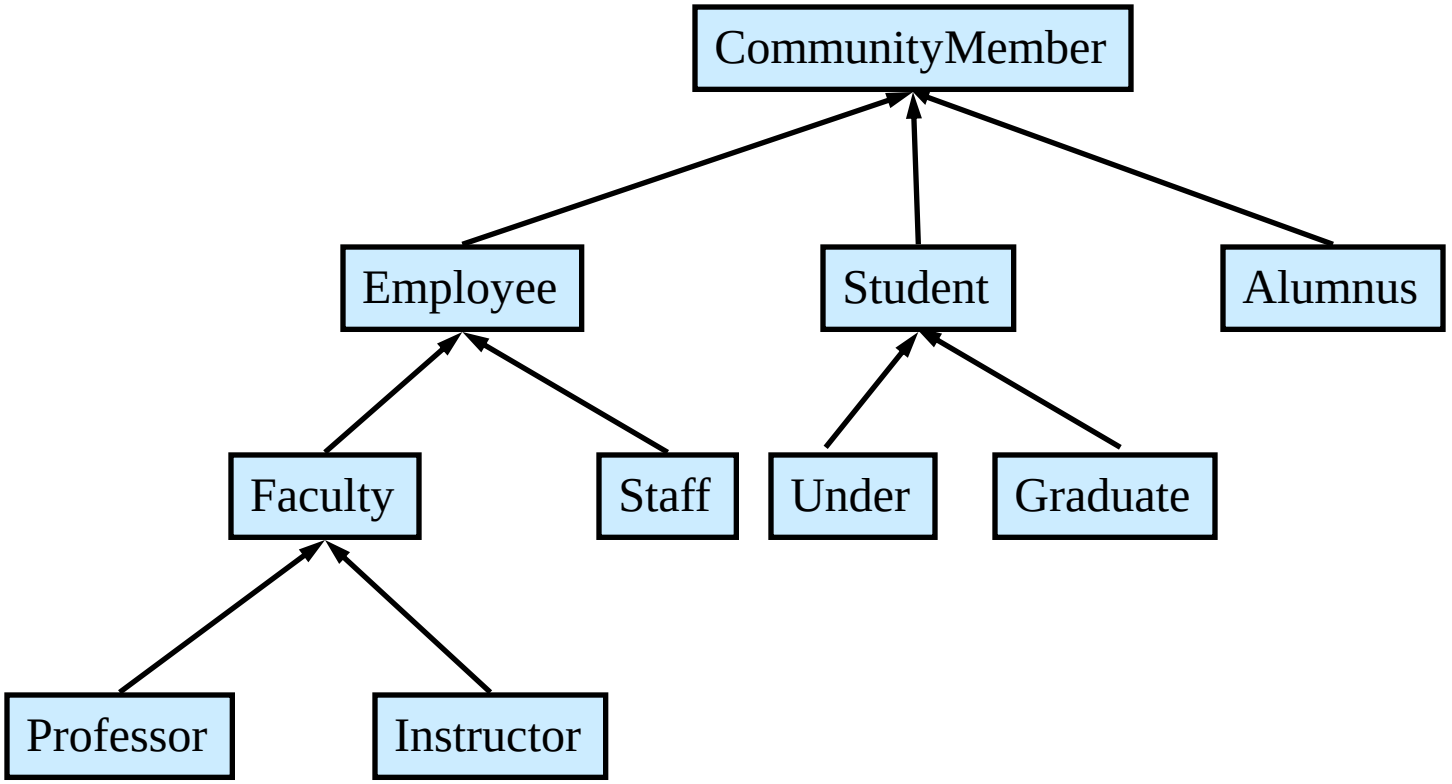
- A child class can *override* the definition of an inherited method in favor of its own
- That is, a child can redefine a method that it inherits from its parent
- The new method must have the same signature as the parent's method, but can have a different implementation.
- The type of the object executing the method determines which version of the method is invoked.

Class Hierarchies

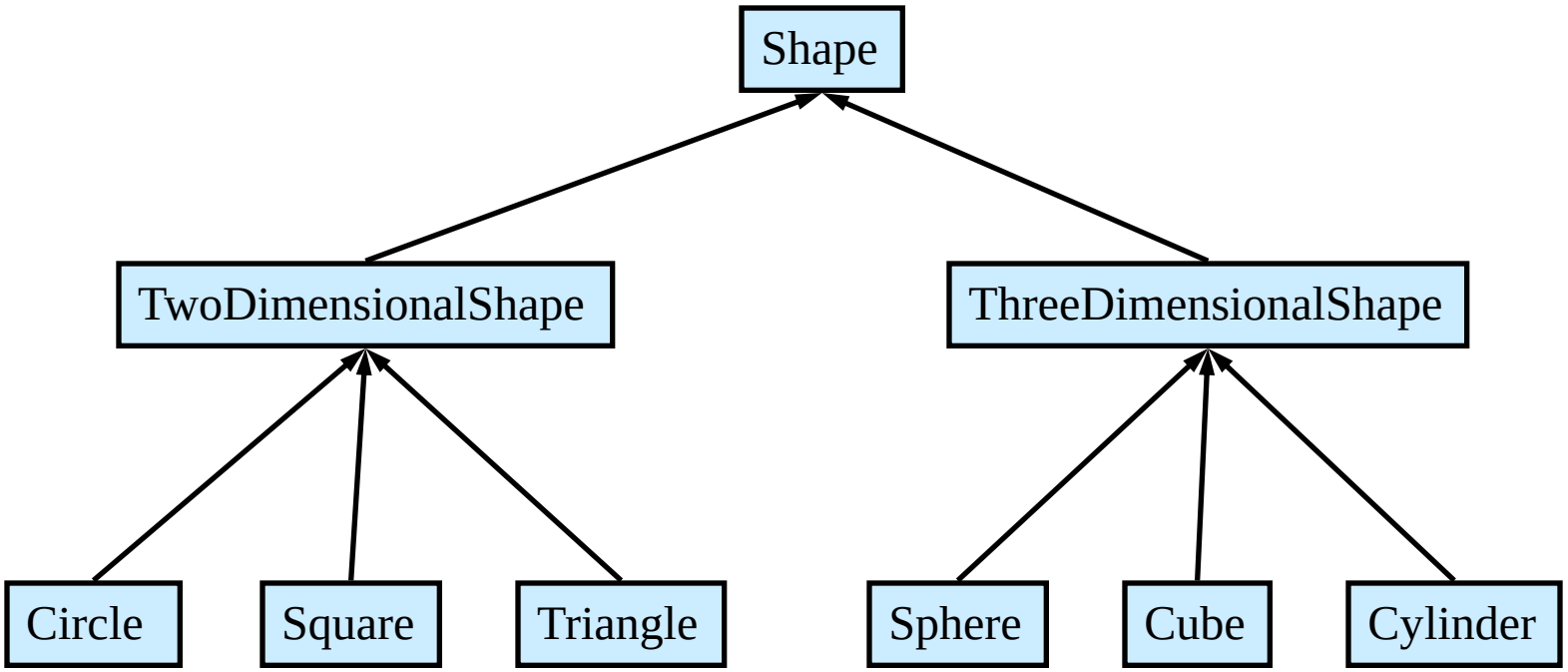
- A child class of one parent can be the parent of another child, forming a *class hierarchy*



Class Hierarchies



Class Hierarchies



Class Hierarchies

- An inherited member is continually passed down the line
  - Inheritance is transitive.
- Good class design puts all common features as high in the hierarchy as is reasonable. Avoids redundant code.

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class derived from it by inheritance.
- For example, if the `Holiday` class is used to derive a child class called `Christmas`, then a `Holiday` reference can be used to point to a `Christmas` object.

```
Holiday day;
day = new Holiday();
…
day = new Christmas();
```

Dynamic Binding

- A polymorphic reference is one which can refer to different types of objects at different times. It morphs!
- The type of the actual instance, not the declared type, determines which method is invoked.
- Polymorphic references are therefore resolved at *run-time*, not during compilation.
  - This is called ***dynamic binding.***

Dynamic Binding

- Suppose the `Holiday` class has a method called `Celebrate`, and the `Christmas` class redefines it (overrides it).
- Now consider the following invocation:

```
day.Celebrate();
```

- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `Celebrate`; if it refers to a `Christmas` object, it invokes the `Christmas` version

Overriding Methods

- C# requires that all class definitions communicate clearly their intentions.
- The keywords *virtual*, *override* and *new* provide this communication.
- If a base class method is going to be overridden it should be declared *virtual.*
- A derived class would then indicate that it indeed does override the method with the *override* keyword.

Overriding Methods

- If a derived class wishes to hide a method in the parent class, it will use the *new* keyword.
- This should be avoided.

Overloading vs. Overriding

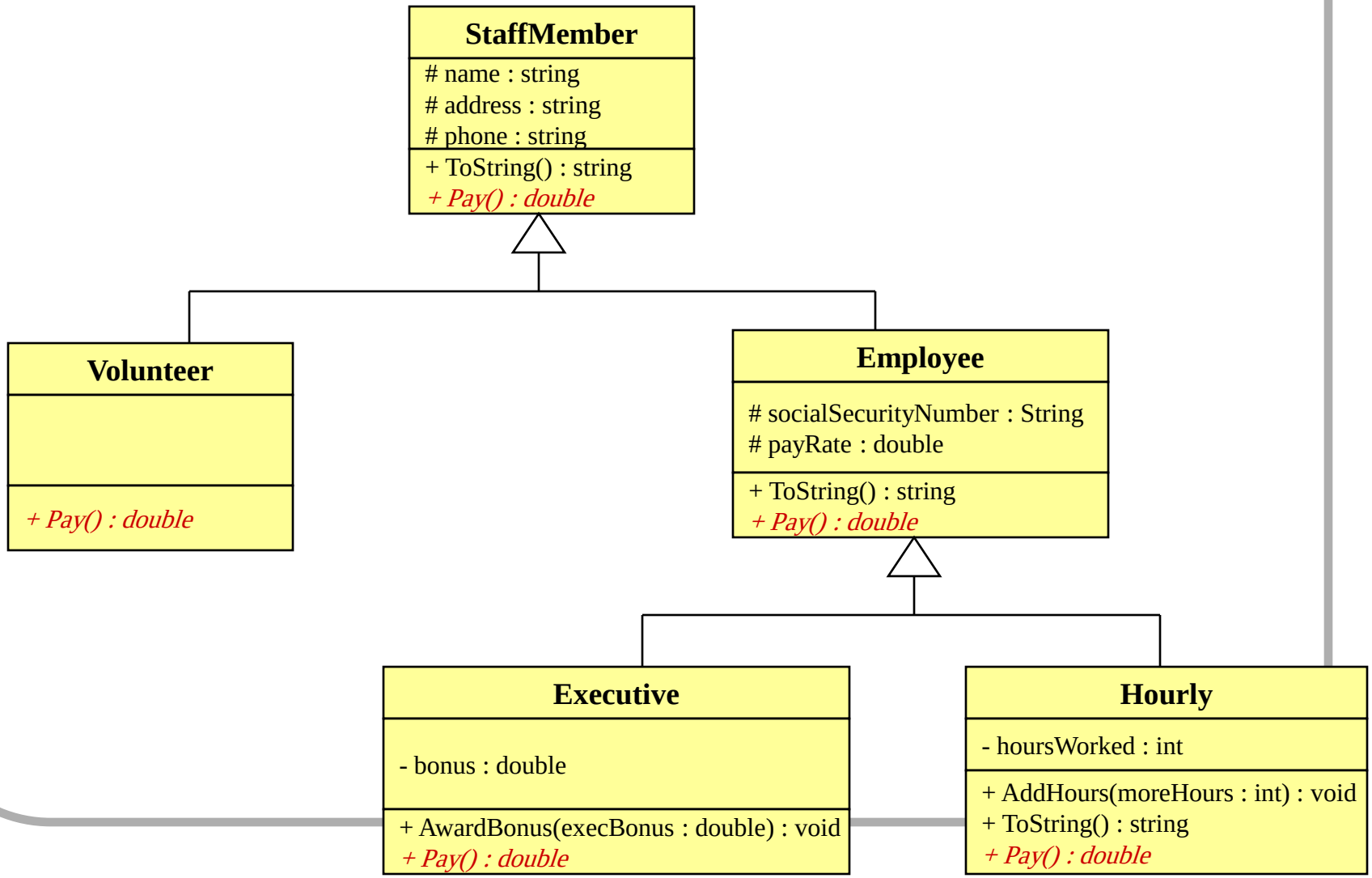
- Overloading deals with multiple methods in the same class with the same name but different signatures
- Overloading lets you define a similar operation in different ways for different data
- Example:

```
int foo(string[] bar);
int foo(int bar1, float a);
```

- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overriding lets you define a similar operation in different ways for different object types
- Example:

```
class Base {
    public virtual int foo() {} }
class Derived {
    public override int foo() {}}
```

Polymorphism via Inheritance



Widening and Narrowing

- Assigning an object to an ancestor reference is considered to be a ***widening*** conversion, and can be performed by simple assignment

```
Holiday day = new Christmas();
```

- Assigning an ancestor object to a reference can also be done, but it is considered to be a ***narrowing*** conversion and must be done with a cast:

```
Christmas christ = new Christmas();
Holiday day = christ;
Christmas christ2 = (Christmas)day;
```

Widening and Narrowing

- Widening conversions are most common.
  - Used in polymorphism.
- Note: Do not be confused with the term widening or narrowing and memory. Many books use *short* to *long* as a widening conversion. A *long* just happens to take-up more memory in this case.
- More accurately, think in terms of sets:
  - The set of animals is greater than the set of parrots.
  - The set of whole numbers between 0-65535 (ushort) is greater (wider) than those from 0-255 (byte).

Type Unification

- **Everything** in C# inherits from `object`
  - Similar to Java except includes value types.
  - Value types are still light-weight and handled specially by the CLI/CLR.
  - This provides a single base type for all instances of all types.
    - Called **Type Unification**

The `System.Object` Class

- All classes in C# are derived from the `Object` class
  - if a class is not explicitly defined to be the child of an existing class, it is a direct descendant of the `Object` class
- The `Object` class is therefore the ultimate root of all class hierarchies.
- The `Object` class defines methods that will be shared by all objects in C#, e.g.,
  - `ToString`: converts an object to a string representation
  - `Equals`: checks if two objects are the same
  - `GetType`: returns the type of a type of object
- A class can override a method defined in Object to have a different behavior, e.g.,
  - String class overrides the `Equals` method to compare the content of two strings