Object Oriented Programming Concepts

Lecture Seven

Polymorphism

(c) 2011. David W. White
School of Computing and Information Technology
Faculty of Engineering and Computing
University of Technology, Jamaica

Email: dwwhite@utech.edu.jm

Object Oriented Programming Concepts

Expected Outcome

At the end of this lecture the student should be able to:

- Understand and be able to implement sub-type polymorphism in C++ and Java
- Understand how an inheritance hierarchy, dynamic binding and overridden methods facilitate sb-type polymorphism
- Use classes in an inheritance hierarchy to demonstrate polymorphism

Object Oriented Programming Concepts

Topics to be covered in this lecture:

- Types of polymorphism
- Liskov substitution principle
- Common methods
- Dynamic binding vs. static binding
- Sample code demonstrating polymorphism
- Polymorphism with abstract methods and classes
- Polymorphism where parent class object is an argument to a method

Polymorphism

- The English word emanates from Greek roots:
  - **Poly** – means "**many**"
  - **Morph** – means "**forms**"

- In Computer Science there are different types of Polymorphism:
  - **Sub-Type Polymorphism**
  - **Ad-Hoc Polymorphism**
  - **Parametric Polymorphism**

Polymorphism

Different types of Polymorphism:

- **Sub-Type Polymorphism**

  Objects of superclass can be replaced by objects of subclass which have common methods

- **Ad-Hoc Polymorphism**

  Methods or functions with the same name exist in different forms and are invoked based on the arguments and return types involved in the call

- **Parametric Polymorphism**

  Involves generic programming, where placeholders or generic data types are substituted by actual types

Polymorphism

Different types of Polymorphism:

- **Sub-Type Polymorphism**

  Objects of a parent class can be substituted by objects of child classes in an inheritance hierarchy

- **Ad-Hoc Polymorphism**

  Method and function **overloading**

- **Parametric Polymorphism**

  Templates in C++ and **type erasure** in Java

Polymorphism

- In this lecture, focus will be on **sub-type polymorphism** which in OOP is often just referred to as simply **polymorphism**
- Polymorphism – ability of a superclass reference to change from referring to superclass objects to subclass objects
- Enables programming in the general
- Eliminates multiple ifs and/or switch statements
- Allows extensibility in programs

Polymorphism

- Employs the Liskov's substitution principle
- Involves a method in the parent class that is also present in, or common to the child classes
- This common method can be overridden by the child classes to provide specialized behaviour for each class as is necessary
- The common method also means the same message can be sent to each object of the various classes involved
- The method is dynamically bound to each object at runtime

Liskov substitution principle

- Objects of a subclass can be substituted wherever objects of the superclass are being used
- Messages sent to the superclass object can be sent to the substituted subclass object without breaking the program
- Guarantees semantic interoperability between objects of classes in an inheritance hierarchy
- Strong behavioral subtyping

Criteria method signatures must meet based on Liskov substitution principle

- Contravariance of method arguments in the subtype.
- Covariance of return types in the subtype.
- No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype.

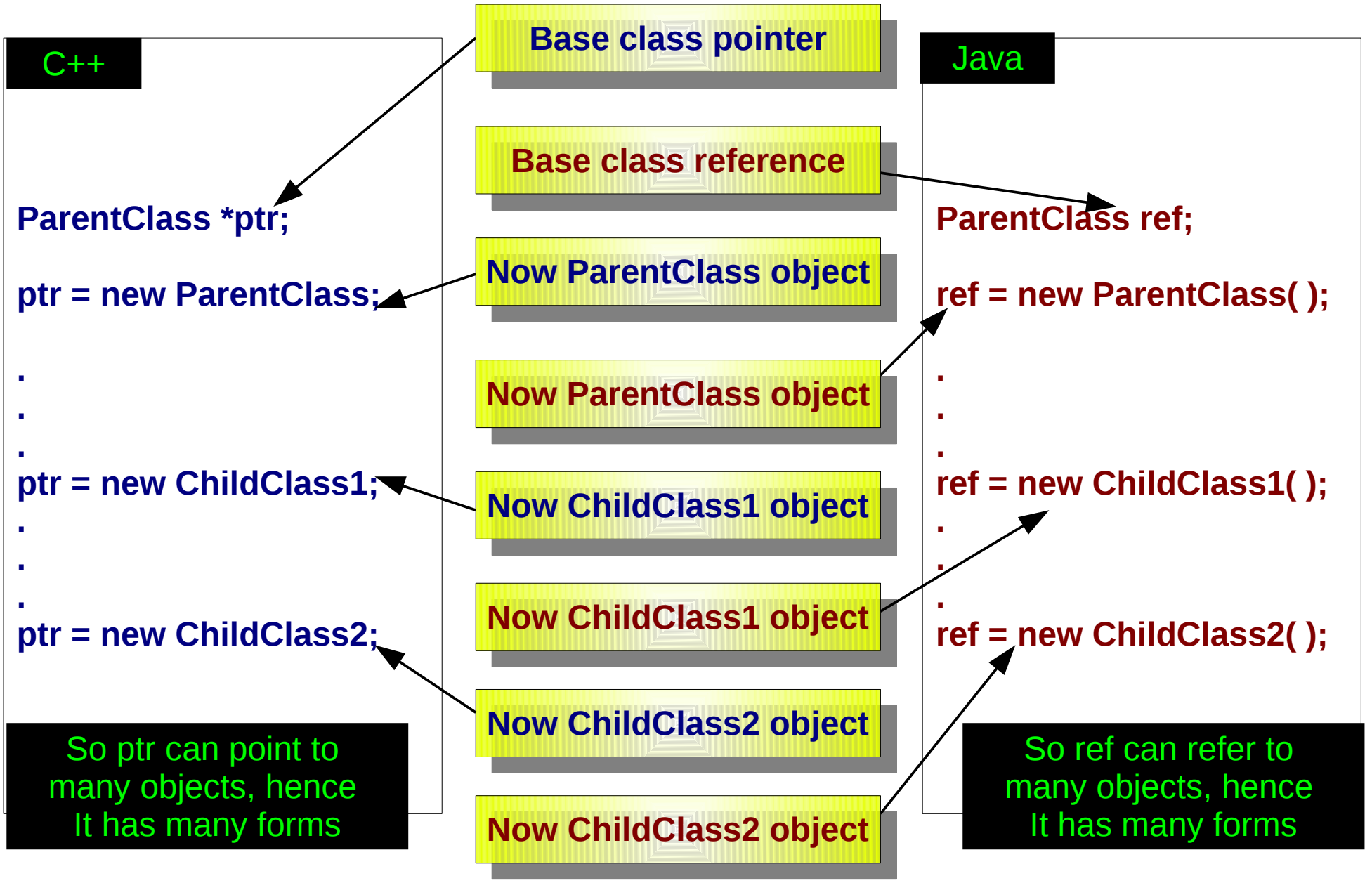
Other criteria method signatures must meet

- Preconditions cannot be strengthened in a subtype.
- Postconditions cannot be weakened in a subtype.
- Invariants of the supertype must be preserved in a subtype.
- History constraint (the "history rule").

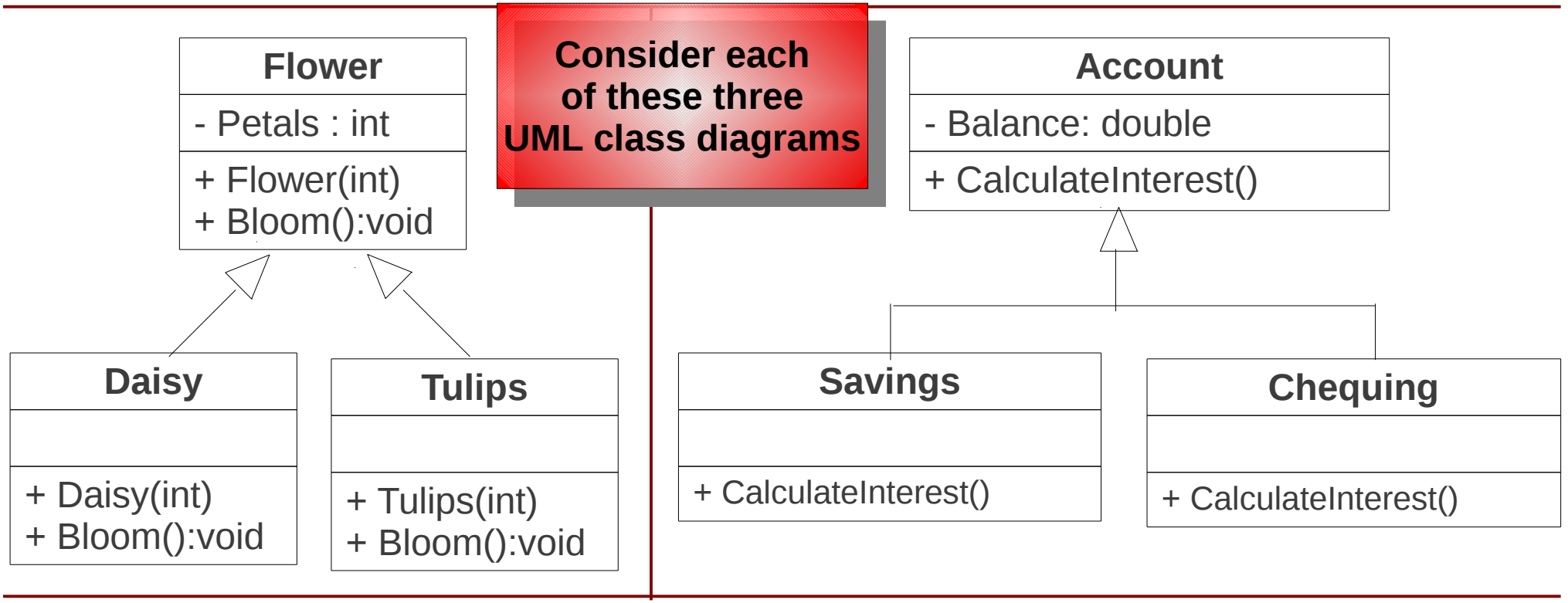
Implementing polymorphism in OOP Languages

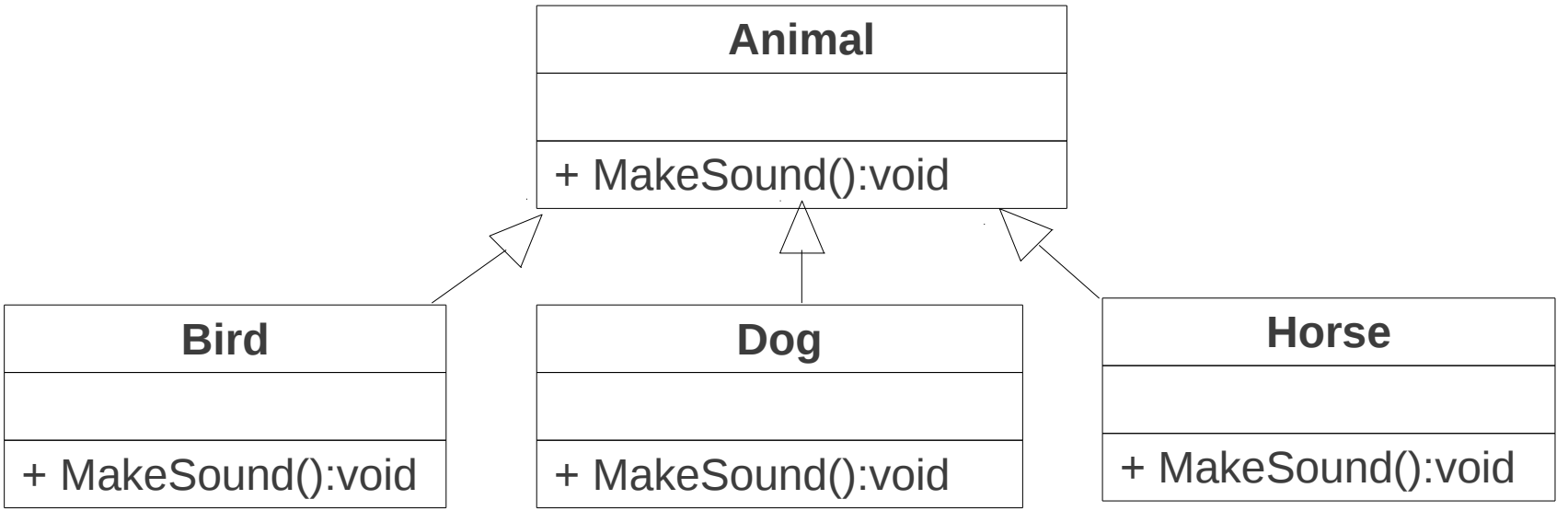
- For polymorphism to work, a pointer or reference is needed of the type of the base class
- This pointer or reference (depending on the language being used) can then point or refer to objects of either the base class or any of its derived classes
- Hence, the pointer or reference can have or take on “many forms”

Implementing polymorphism in OOP Languages

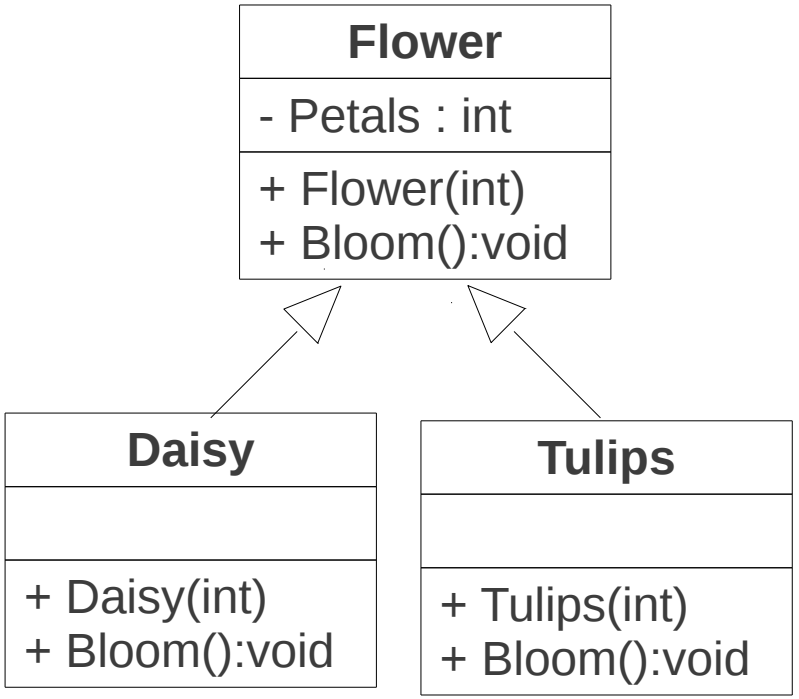


The Role of Common Methods in Polymorphism





The Role of Common Methods in Polymorphism



Bloom( ) is a method that is common to all three classes so objects of the parent class can be replaced by objects of the child class and the call to Bloom( ) will still work. The appropriate version of Bloom( ) will be invoked.

C++
X->Bloom( )

Java
X.Bloom( );

can be replaced by

C++

Parent Class

```
Flower *Xptr;
Xptr = new Flower;
Xptr->Bloom( );
```

C++

Child Class

```
Flower *Xptr;
Xptr = new Daisy;
Xptr->Bloom( );
```

can be replaced by

Java

Parent Class

```
Flower Xref;
Xref = new Flower();
Xref.Bloom( );
```

Java

Child Class

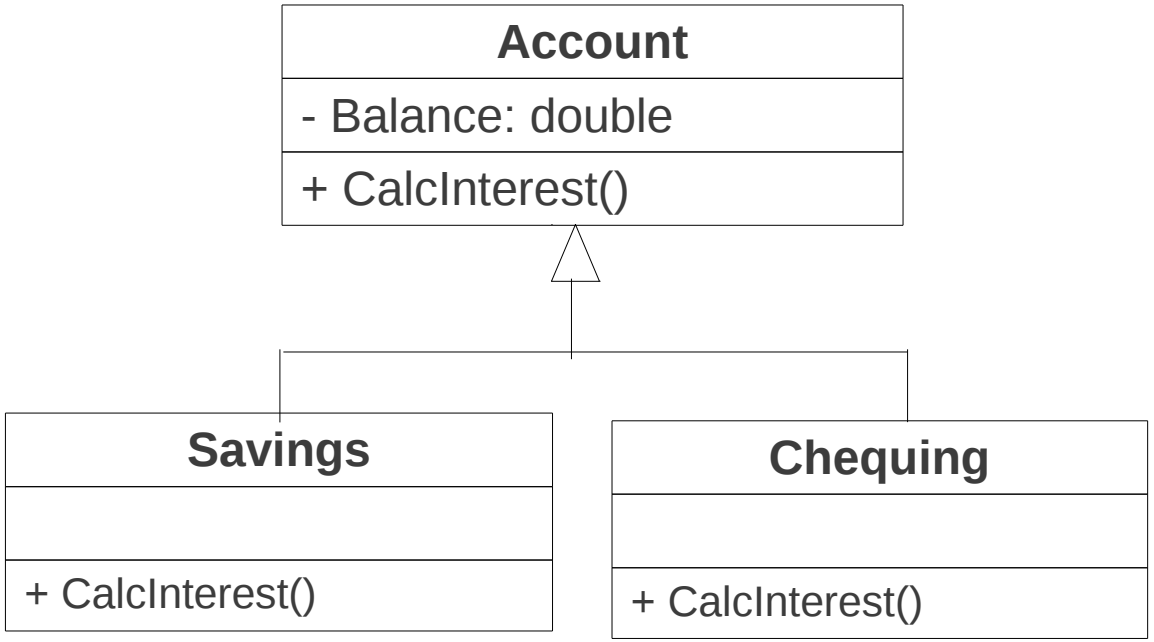
```
Flower Xref;
Xref = new Daisy();
Xref.Bloom( );
```

The Role of Common Methods in Polymorphism

CalcInterest( ) is a method that is common to all three classes so objects of the parent class can be replaced by objects of the child class and the call to bloom will still work. The appropriate version of CalcInterest( ) will be invoked.

C++
Obj->CalcInterest( )

Java
Obj.CalcInterest( );



can be replaced by

C++

Parent Class

```
Account *oPtr;
oPtr = new Account;
oPtr->CalcInterest( );
```

C++

Child Class

```
Account *oPtr;
oPtr = new Chequing;
oPtr->CalcInterest( );
```

can be replaced by

Java

Parent Class

```
Account oRef;
oRef = new Account();
oRef.CalcInterest( );
```

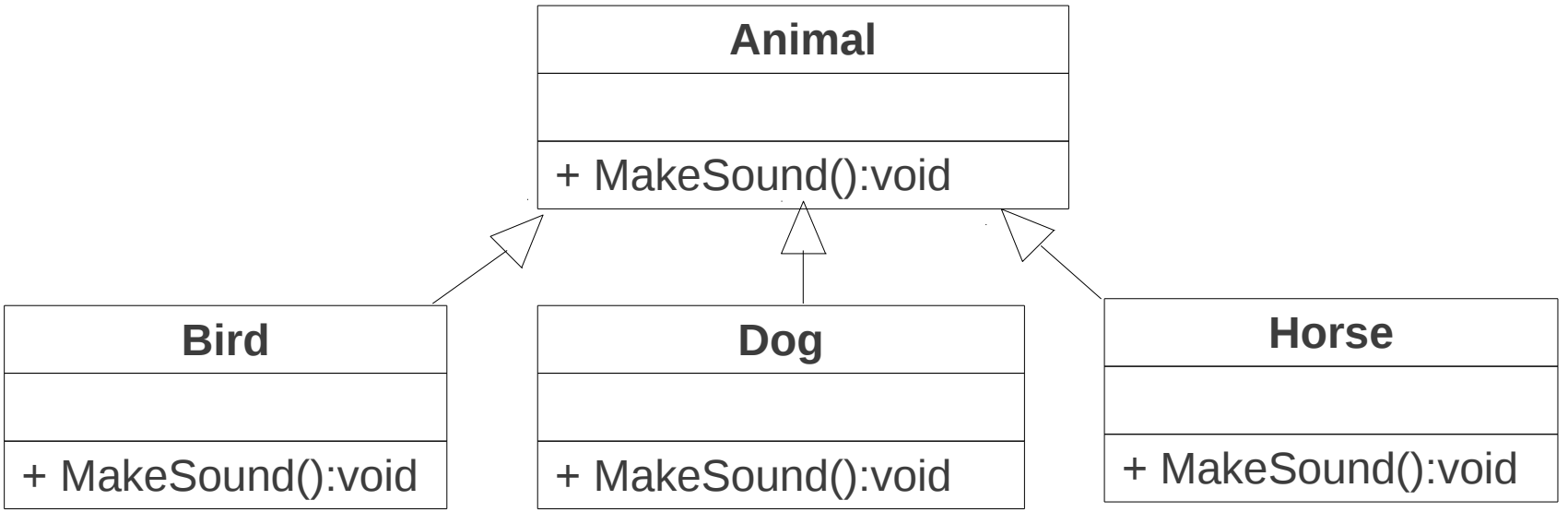
Java

Child Class

```
Account oRef;
oRef = new Chequing();
oRef.CalcInterest( );
```

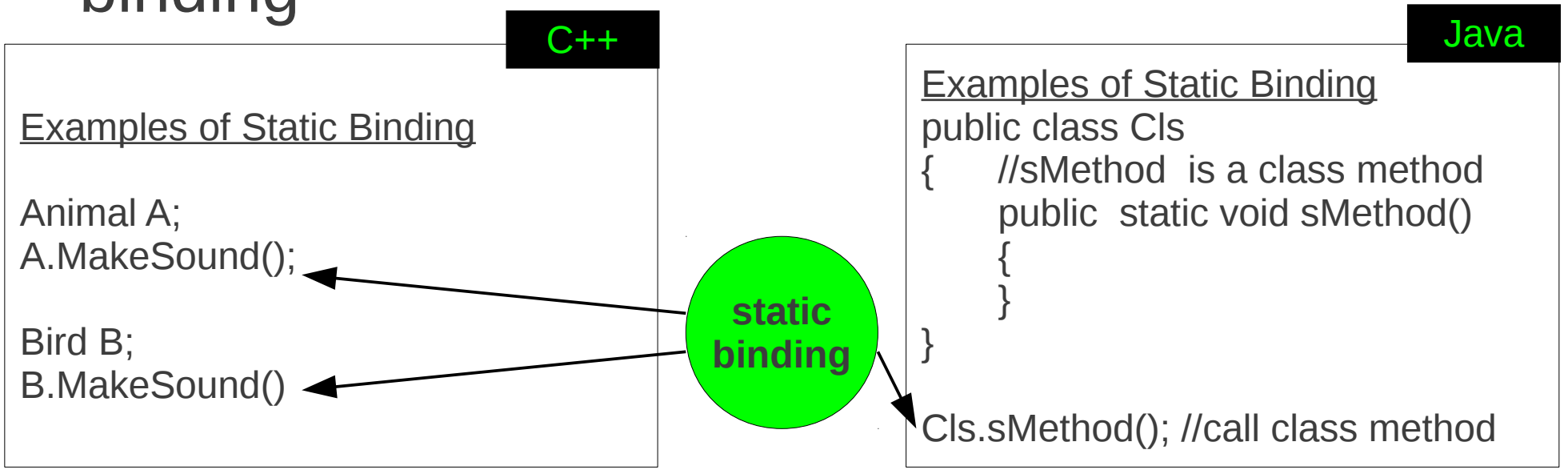
The Role of Common Methods in Polymorphism

Examine the UML class diagram below very carefully. Can you identify any parent class, child classes and common methods? Write code using all four classes to demonstrate polymorphism at work. See the answer in the Sample Code section of this lecture on later slides.



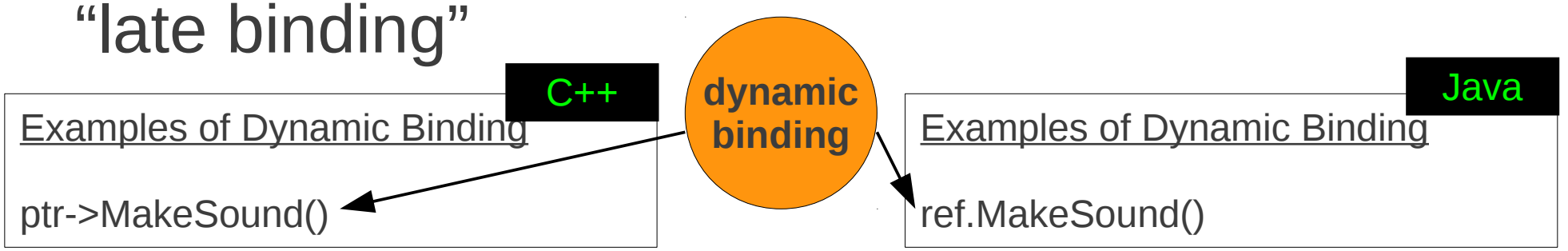
Dynamic Binding vs. Static Binding

- Static binding is done at compile time
- This is because the compiler can determine which actual method to associate the method call with while the program is being compiled
- Therefore, static binding is also called “early binding”



Dynamic Binding vs. Static Binding

- Dynamic binding has to be done at runtime
- This is because the program cannot determine which actual method to associate the method call with until it reaches the line invoking the call
- It then examines the object variable to see which object it currently holds, and thus invokes the method from that objects' class
- For this reason, dynamic binding is also called “late binding”



Dynamic Binding vs. Static Binding

- Recall that methods can be overridden in an inheritance hierarchy, and
- Parent class objects can be substituted by child class objects.
- This makes it difficult for the compiler to know which version of a method is to be invoked when it see a message being sent to an object in an inheritance hierarchy
- Therefore polymorphism employs dynamic binding to resolve method calls

Dynamic Binding vs. Static Binding

Static Binding in C++

```
Animal animalObject;
AnimalObject.MakeSound( );

Bird birdObject;
birdObject.MakeSound( );

Dog  dogObject;
dogObject.MakeSound( );

Horse  horseObject;
horseObject.MakeSound( );
```

AnimalObject will always be an object of the class Animal in this scope, so the compiler can know easily that it should invoke MakeSound( ) from the Animal class, and not another class.

The same is true for all the other three classes. The compiler will have no trouble determining which MakeSound( ) method to call – it will invoke the appropriate method for each object in this example, and can determine this at runtime.

Dynamic Binding vs. Static Binding

Dynamic Binding in C++

```
Animal *objectPointer;
objectPointer = new Animal;
objectPointer->MakeSound( );

objectPointer = new Bird;
objectPointer->MakeSound( );

objectPointer = new Dog;
objectPointer->MakeSound( );

objectPointer = new Horse;
objectPointer->MakeSound( );
```

Here ObjectPointer is a pointer, and there is no telling what object it might be pointing to at any given momemt, so the program has to wait until runtime to determine which object objectPointer is pointing to and thus invoke the corresponding MakeSound( ) for that object

Here we have the same pointer, but as we can see it can point to a number of different objects, so the call to MakeSound( ) cannot be resolved until we know what the pointer is pointing to, and we cant know that until the program is actually running

Dynamic Binding vs. Static Binding

Dynamic Binding in Java

```
Animal objectReference;
objectReference = new Animal();
ObjectReference.MakeSound( );


objectReference = new Bird();
ObjectReference.MakeSound( );


objectReference = new Dog();
ObjectReference.MakeSound( );


objectReference = new Horse();
ObjectReference.MakeSound( );
```

Here objectReference is a reference, and there is no telling what object it might be refering to at any given momemt, so the program has to wait until runtime to determine which object objectReference is refering to and thus invoke the corresponding MakeSound( ) for that object

Here we have the same pointer, but as we can see it can refer to a number of different objects, so the call to MakeSound( ) cannot be resolved until we know what the reference is refering to, and we cant know that until the program is actually running

Dynamic Binding vs. Static Binding

- Consider the following lines of code:

```
for (j=0;  j < Max; j++)                    //C++
    AnimalArray[j]->MakeSound();

for (j=0;  j < Max; j++)                    //Java
    AnimalArray[j].MakeSound();
```

- The compiler has no way of knowing which version of MakeSound( ) to call on each iteration of the for loop, since polymorphism allows each array element to refer to either a parent or child class object.

Sample Code Demonstrating Polymorphism

C++

```
//AnimalDriver to demonstrate polymorphism
#include "Animal.h"
#include "Bird.h"
#include "Dog.h"
#include "Horse.h"
int main()
{
	//object pointer variable can point
	//to objects of parent class Animal
	//and its child classes
	Animal *obj;

	obj = new Animal;
	obj->MakeSound();

	obj = new Bird;
	obj->MakeSound();

	obj = new Dog;
	obj->MakeSound();

	obj = new Horse;
	obj->MakeSound();
	return 0;
}
```

Java

```
//AnimalDriver to demonstrate polymorphism
public class AnimalDriver
{
	public static void main(String[ ] args)
	{
		//object variable can reference
		// parent class Animal
		//and its child classes
		Animal obj;

		obj = new Animal();
		obj.MakeSound();

		obj = new Bird();
		obj.MakeSound();

		obj = new Dog();
		obj.MakeSound();

		obj = new Horse();
		obj.MakeSound();
	}
}
```

Sample Code – Polymorphism using array of objects

C++

```
//AnimalDriver to demonstrate polymorphism
#include "Animal.h"
#include "Bird.h"
#include "Dog.h"
#include "Horse.h"

int main()
{
	//each element of array can point
	// to objects of parent class Animal
	//and objects of its child classes
	Animal *AnimalArray[4];
	AnimalArray[0] = new Animal;
	AnimalArray[1] = new Bird;
	AnimalArray[2] = new Dog;
	AnimalArray[3] = new Horse;

	for (int count = 0; count < 4; ++count)
	{
		AnimalArray[count]->MakeSound();
	}
	return 0;
}
```

Java

```
//AnimalDriver to demonstrate polymorphism
public class AnimalDriver
{
	public static void main(String[ ] args)
	{
		//each element of array can reference
		// objects of parent class Animal
		//and objects of its child classes
		Animal [ ] AnimalArray = new Animal[4];
		AnimalArray[0] = new Animal();
		AnimalArray[1] = new Bird();
		AnimalArray[2] = new Dog();
		AnimalArray[3] = new Horse();

		for (int count = 0; count < 4; ++count)
		{
			AnimalArray[count].MakeSound();
		}
	}
}
```

Sample Code – Parent Class Animal

C++

```
//Animal class
#ifndef AnimalCl
#define AnimalCl
#include <iostream>
using namespace std;
class Animal
{
	public:
	virtual void MakeSound()
	{
		cout <<
		"MakeSound() in Animal" << endl;
	}
};
#endif
```

Java

```
//Animal class
public class Animal
{
	public void MakeSound()
	{
		System.out.println(
		"MakeSound() in Animal");
	}
}
```

Sample Code – Child Class Bird

C++

```
#ifndef BirdCl
#define BirdCl
#include "Animal.h"
#include <iostream>
using namespace std;
class Bird : public Animal
{
	public:
	void MakeSound()
	{
		cout <<
		"MakeSound() in Bird" << endl;
	}
};
#endif
```

Java

```
public class Bird extends Animal
{
	public void MakeSound()
	{
		System.out.println(
		"MakeSound() in Bird");
	}
}
```

Sample Code – Child Class Dog

C++

```
#ifndef DogCl
#define DogCl
#include "Animal.h"
#include <iostream>
using namespace std;
class Dog : public Animal
{
	public:
	void MakeSound()
	{
		cout <<
		"MakeSound() in Dog" << endl;
	}
};
#endif
```

Java

```
public class Dog extends Animal
{
	public void MakeSound()
	{
		System.out.println(
		"MakeSound() in Dog");
	}
}
```

Sample Code – Child Class Horse

C++

```
#ifndef HorseCl
#define HorseCl
#include "Animal.h"
#include <iostream>
using namespace std;
class Horse : public Animal
{
	public:
	void MakeSound()
	{
		cout <<
		"MakeSound() in Horse" << endl;
	}
};
#endif
```

Java

```
public class Horse extends Animal
{
	public void MakeSound()
	{
		System.out.println(
		"MakeSound() in Horse");
	}
}
```

Polymorphism with Overridden Methods

- In C++, note that unless the base class method is declared with the keyword **virtual**, the call to the subclass methods in the previous examples will not work – no error will be generated, but only the base class method will be invoked
- In Java, you dont have to declare the base class method with any other keyword, the code in the previous examples will just work.
- However Java provides the optional annotation **@override** in the sub class which indicates that the subclass method overrides a superclass method. An error is generated if this is not true.

Polymorphism with Overridden Methods

- Example of use of keyword **virtual** in C++

```
//Animal class
#ifndef AnimalCl
#define AnimalCl
#include <iostream>
using namespace std;
class Animal
{
	public:
	virtual void MakeSound()
	{
		cout << "MakeSound() in Animal" << endl;
	}
};
#endif
```

Polymorphism with Overridden Methods

- Example of use of annotation **@override** in Java

```
public class Bird extends Animal
{
	@Override
	public void MakeSound()
	{
		System.out.println("MakeSound() in Bird");
	}
}
```

- Note – you cannot override any method declared as **final** in a super class in Java

Polymorphism with Abstract Methods and Classes

- As discussed in lecture six – Inheritance Part Two, an abstract class is a class with one or more methods for which no implementation is provided
- Any child class inheriting from a abstract parent class itself becomes abstract unless it implements all the parent's abstract methods
- You cannot instantiate objects of an abstract class
- You can use an abstract class to force child classes to implement the parent's methods

Polymorphism with Abstract Methods and Classes

- For example, if we make the MakeSound() method in the parent class Animal abstract, then we can no longer create objects of Animal, but we can declare pointers and references to Animal
- The child classes remain concrete because they all (Bird, Dog, Horse) implement their own version of MakeSound()
- However, the Driver file has to change to remove the line where we were previously creating objects of the Animal class

Sample Code – Modified Abstract Parent Class Animal

C++

```
//Animal class
#ifndef AnimalCl
#define AnimalCl
#include <iostream>
using namespace std;
class Animal
{
	public:
	virtual void MakeSound() = 0;
};
#endif
```

Java

```
//Animal class
public abstract class Animal
{
	public abstract void MakeSound();
}
```

The implementation for the MakeSound() method has been removed from the Animal class and therefore the Animal class now becomes abstract.

In C++ precede the method return type by keyword virtual and end with = 0;

In Java add keyword abstract before return type of method, and add abstract before keyword class too.

Sample Code – Modified Driver for Abstract Parent

C++

```
//AnimalDriver to demonstrate polymorphism
#include "Animal.h"
#include "Bird.h"
#include "Dog.h"
#include "Horse.h"
int main()
{
	//object pointer variable can point
	//to objects of parent class Animal
	//and its child classes
	Animal *obj;

	//obj = new Animal;
	//obj->MakeSound();

	obj = new Bird;
	obj->MakeSound();

	obj = new Dog;
	obj->MakeSound();

	obj = new Horse;
	obj->MakeSound();
	return 0;
}
```

Java

```
//AnimalDriver to demonstrate polymorphism
public class AnimalDriver
{
	public static void main(String[ ] args)
	{
		//object variable can reference
		// parent class Animal
		//and its child classes
		Animal obj;

		//obj = new Animal();
		//obj.MakeSound();

		obj = new Bird();
		obj.MakeSound();

		obj = new Dog();
		obj.MakeSound();

		obj = new Horse();
		obj.MakeSound();
	}
}
```

Sample Code – Modified Driver for Abstract Parent

C++

```
//AnimalDriver to demonstrate polymorphism
#include "Animal.h"
#include "Bird.h"
#include "Dog.h"
#include "Horse.h"
int main()
{
	//object pointer variable can point
	//to objects of parent class Animal
	//and its child classes
	Animal *obj;

	//obj = new Animal;
	//obj->MakeSound();

	obj = new Bird;
	obj->MakeSound();

	obj->MakeSound();
	return 0;
}
```

Java

```
//AnimalDriver to demonstrate polymorphism
public class AnimalDriver
{
	public static void main(String[ ] args)
	{
		//object variable can reference
		// parent class Animal
		//and its child classes
		Animal obj;

		//obj = new Animal();
		//obj.MakeSound();

		obj = new Bird();
		obj.MakeSound();

		obj = new Dog();

}
```

The lines in red above have to be commented out or removed as they will no longer compile without error. Since the Animal class is now abstract, we can no longer create or instantiate objects of the class Animal.

Polymorphism in cases where a method expects a parent class as a parameter

- If the argument to a method is an object of a parent class in an inheritance hierarchy
- All parent class objects can be safely substituted by child class objects in the method call
- Polymorphism will be achieved when objects other than just those of the parent class are passed into the function, as long as the objects passed are child classes of the parent

Sample Code – Using method parameter for polymoprhism

C++

```
#include "Animal.h"
#include "Bird.h"
#include "Dog.h"
#include "Horse.h"

void ProcessSound(Animal *oPtr)
{
	oPtr->MakeSound();
}

int main()
{
	Animal *obj;

	obj = new Bird;
	ProcessSound(obj);

	obj = new Dog;
	ProcessSound(obj);

	obj = new Horse;
	ProcessSound(obj);

	return 0;
}
```

Java

```
//AnimalDriver to demonstrate polymorphism
public class AnimalDriver
{
	public static void ProcessSound(Animal oRef)
	{
		oRef.MakeSound();
	}

	public static void main(String[] args)
	{
		Animal obj;

		obj = new Bird();
		ProcessSound(obj);

		obj = new Dog();
		ProcessSound(obj);

		obj = new Horse();
		ProcessSound(obj);
	}
}
```