Object-Oriented Software Engineering
Practical Software Development using UML and Java

Chapter 2:
Review of Object Orientation

2.1 What is Object Orientation?

Procedural paradigm:

- Software is organized around the notion of *procedures*
- *Procedural abstraction*
  - —Works as long as the data is simple

Adding *data abstractions* groups together the pieces of data that describe some entity

- —Helps reduce the system's complexity.
  - Such as *Records* and *structures*

Object oriented paradigm:

- Organizing procedural abstractions in the context of data abstractions

Here, the program is centered around procedures, while data is kept simple in arrays.

```c
// C-style procedural example
#include <stdio.h>

// Simple data: just two arrays
char studentNames[3][20] = {"Alice", "Bob", "Charlie"};
int studentGrades[3] = {90, 85, 78};

// Procedure to print student info
void printStudents() {
    for (int i = 0; i < 3; i++) {
        printf("Name: %s, Grade: %d\n", studentNames[i], studentGrades[i]);
    }
}

int main() {
    printStudents(); // System organized around functions
    return 0;
}
```

Now we group related data (name + grade) into a structure. This makes it easier to manage and reduces complexity.

```c
typedef struct {
    char name[20];
    int grade;
} Student;

Student students[3] = {
    {"Alice", 90},
    {"Bob", 85},
    {"Charlie", 78}
};

// Procedure works with data abstraction
void printStudents(Student arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("Name: %s, Grade: %d\n", arr[i].name, arr[i].grade);
    }
}

int main() {
    printStudents(students, 3);
    return 0;
}
```

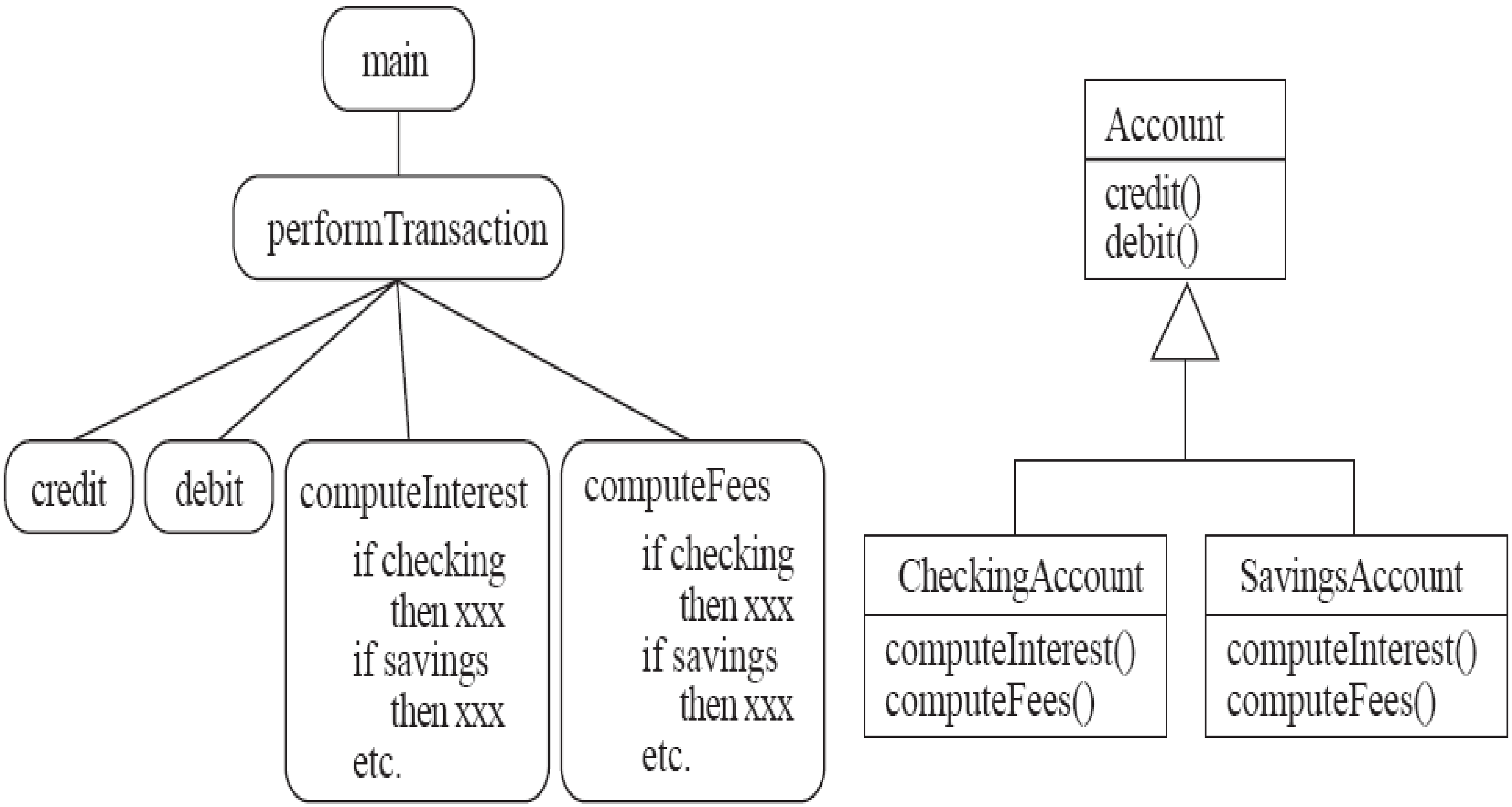
```java
// Java OOP example
class Student {
    String name;
    int grade;

    // Constructor
    Student(String name, int grade) {
        this.name = name;
        this.grade = grade;
    }

    // Behavior is tied to the object
    void printInfo() {
        System.out.println("Name: " + name + ", Grade: " + grade);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Alice", 90);
        Student s2 = new Student("Bob", 85);
        Student s3 = new Student("Charlie", 78);

        s1.printInfo();
        s2.printInfo();
        s3.printInfo();
    }
```

Object Oriented paradigm

All computations are performed in the context of objects.

- The objects are instances of classes, which:
  - —are data abstractions
  - —contain procedural abstractions that operate on the objects
- A running program can be seen as a collection of objects collaborating to perform a given task

A View of the Two paradigms



Procedural Paradigm

Pros:

1. **Simplicity** – Easy to implement for small programs with simple data.
2. **Direct control** – Programmer explicitly decides how procedures handle data.
3. **Performance** – Typically faster due to less abstraction (no object overhead).
4. **Good for small projects** – Clear procedural flow, straightforward logic.

Cons:

1. **Scalability issues** – As the system grows, adding new account types requires changing multiple functions (e.g., updating computeInterest and computeFees everywhere).
2. **Duplication** – Conditional checks (if checking, if savings) spread across many functions.
3. **Tight coupling** – Procedures depend on knowledge of data types → makes maintenance harder.
4. **Low extensibility** – To add a new account type (e.g., BusinessAccount), many procedures must be updated manually.

Object-Oriented Paradigm

Pros:
Encapsulation – Each account type manages its own data and behavior, reducing complexity.
Extensibility – Adding a new account type only requires creating a new subclass, not modifying existing code.
Code reuse – Shared behavior is factored into the base Account class.
Maintainability – No need for conditionals like if checking / if savings; polymorphism takes care of it.
Abstraction – Hides unnecessary implementation details, focusing on behavior at the object level.

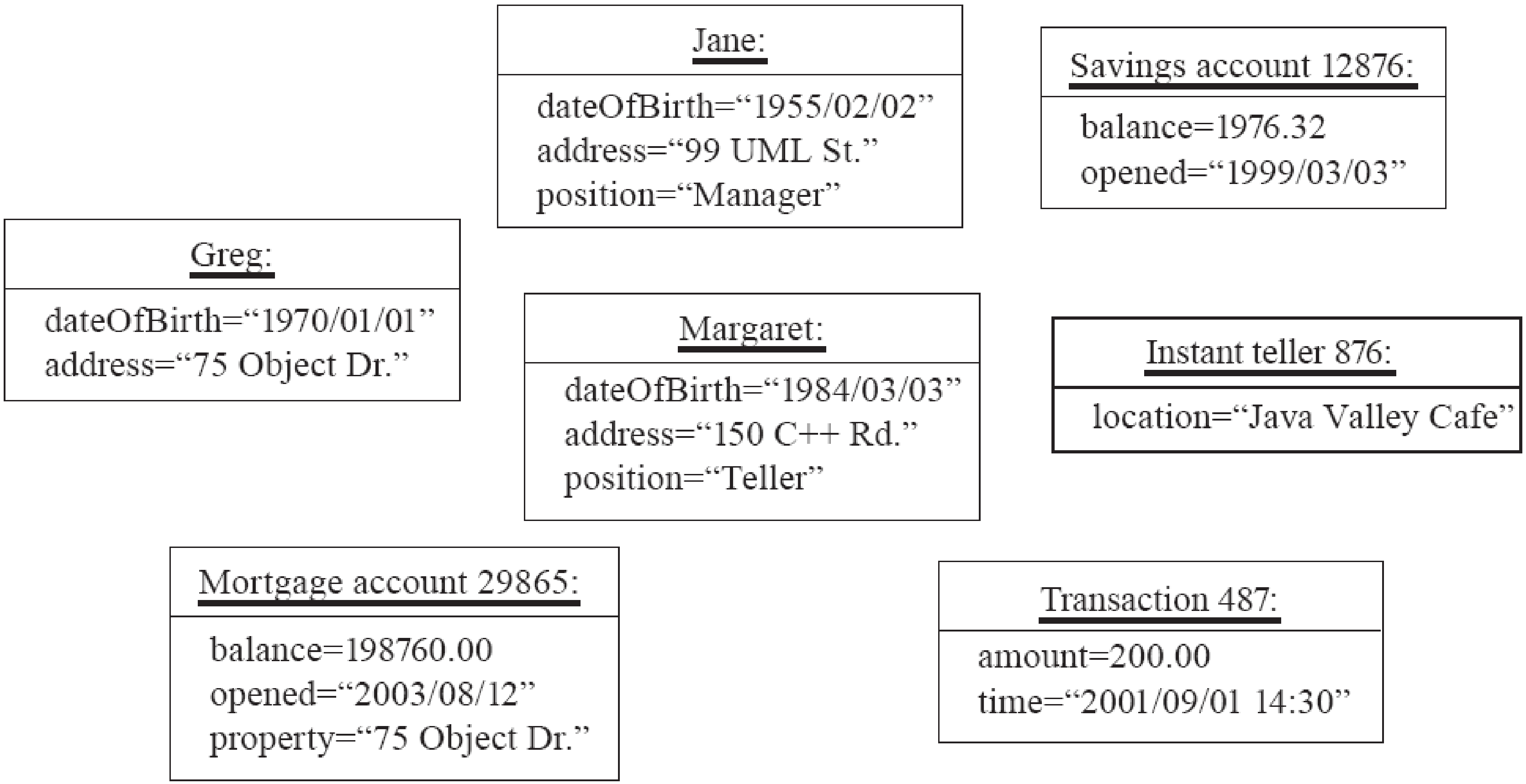
Cons:
Learning curve – More complex concepts (inheritance, polymorphism) compared to procedural programming.
Overhead – Objects, dynamic dispatch, and class hierarchies can be slower than procedural code.
Design complexity – Requires careful planning to avoid bad hierarchies (e.g., deep inheritance chains).
Overkill for small programs – If the system is simple, OOP can add unnecessary complexity.

2.2 Classes and Objects

Object

- A chunk of structured data in a running software system
- Has *properties*
  - —Represent its state
- Has *behaviour*
  - —How it acts and reacts
  - —May simulate the behaviour of an object in the real world

Objects: Shown as a UML instance diagram



www.lloseng.com

Classes

A class:

- A unit of abstraction in an object oriented (OO) program

- Represents similar objects
  —Its *instances*

- A kind of software module
  —Describes its instances' structure (properties)
  —Contains *methods* to implement their behaviour

Is Something a Class or an Instance?

- Something should be a *class* if it could have instances
- Something should be an *instance* if it is clearly a *single* member of the set defined by a class

Film

- Class; instances are individual films.

***Reel of Film*:**

- Class; instances are physical reels

Film reel with serial number SW19876

- Instance of **`ReelOfFilm`**

Science Fiction

- Instance of the class **`Genre`**.

Science Fiction Film

- Class; instances include ‘Star Wars’

***Showing of ‘Star Wars’ in the Phoenix Cinema at 7 p.m*.:**

- Instance of **`ShowingOfFilm`**

Naming classes

- Use *capital* letters
  - —E.g. `BankAccount` not `bankAccount`
- Use *singular* nouns
- Use the right level of generality
  - —E.g. `Municipality`, not `City`
- Make sure the name has only *one* meaning
  - —E.g. ‘bus’ has several meanings

2.3 Instance Variables

Variables defined inside a class corresponding to data present in each instance

- Also called *fields* or *member variables*

- Attributes
  - —Simple data
  - —E.g. `name`, `dateOfBirth`

- Associations
  - —Relationships to other important classes
  - —E.g. `supervisor`, `coursesTaken`
  - —More on these in Chapter 5

www.lloseng.com

Variables vs. Objects

A variable

- *Refers* to an object
- May refer to different objects at different points in time

An object can be referred to by several different variables at the same time

***Type* of a variable**

- Determines what classes of objects it may contain

A variable doesn’t hold the object itself, but rather a reference (pointer) to it.

```
Student s1 = new Student("Alice", 90);
```

A variable may refer to different objects at different points in time

```
Student s1 = new Student("Alice", 90);  // s1 → Alice
s1 = new Student("Bob", 85);            // s1 → Bob
```

An object can be referred to by several different variables at the same time

```
Student s1 = new Student("Alice", 90);
Student s2 = s1;   // both s1 and s2 point to the same object

s2.grade = 95;     // modifying through s2
System.out.println(s1.grade);  // prints 95
```

Type of a variable determines what classes of objects it may contain

```
Student s1;         // can only hold references to Student objects
String name;        // can only hold references to String objects
Object o;           // can hold references to ANY object
```

Class variables

A *class variable's* value is *shared* by all instances of a class.

- Also called a *static* variable

- If one instance sets the value of a class variable, then all the other instances see the same changed value.

- Class variables are useful for:
    - —Default or 'constant' values (e.g. PI)
    - —Lookup tables and similar structures

Caution: *do not over-use class variables*

Also called a static variable it belongs to the class itself, not to individual objects.

```java
class Student {
    String name;
    static int schoolCode = 1234; // class variable
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.name = "Alice";

        Student s2 = new Student();
        s2.name = "Bob";

        // Both s1 and s2 share the same schoolCode
        System.out.println(s1.schoolCode); // 1234
        System.out.println(s2.schoolCode); // 1234
    }
}
```

```java
class Student {
    String name;
    static int schoolCode = 1234; // shared across all students
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student();

        s1.schoolCode = 5678;  // s1 modifies it

        // Both s1 and s2 see the change
        System.out.println(s1.schoolCode); // 5678
        System.out.println(s2.schoolCode); // 5678
    }
}
```

Caution: Do not over-use class variables

If too many variables are static, the class loses flexibility.

```
class Student {
    String name;
    static int grade;  // ❌ Bad design!
}
```

2.4 Methods, Operations and Polymorphism

Operation

- A higher-level procedural abstraction that specifies a type of behaviour

- Independent of any code which implements that behaviour
    - —E.g. calculating area (in general)

Methods, Operations and Polymorphism

Method

- A procedural abstraction used to implement the behaviour of a class

- Several different classes can have methods with the same name
  - —They implement the same abstract operation in ways suitable to each class
  - —E.g. calculating area in a rectangle is done differently from in a circle

Polymorphism

A property of object oriented software by which an *abstract operation may be performed in different ways* in different classes.

- Requires that there be *multiple methods of the same name*
- The choice of which one to execute depends on the object that is in a variable
- Reduces the need for programmers to code many `if-else` or `switch` statements

Each class implements the same method but in its own way.

```
class Account {
    void computeInterest() {
        System.out.println("Generic account interest");
    }
}

class CheckingAccount extends Account {
    @Override
    void computeInterest() {
        System.out.println("Checking account: no interest");
    }
}

class SavingsAccount extends Account {
    @Override
    void computeInterest() {
        System.out.println("Savings account: 3% interest");
    }
}
```

Even though both variables are of type `Account`, the actual object determines which method is executed.

```
public class Main {
    public static void main(String[] args) {
        Account a1 = new CheckingAccount();  // variable type: Account, object: CheckingAccount
        Account a2 = new SavingsAccount();   // variable type: Account, object: SavingsAccount

        a1.computeInterest(); // → Checking account: no interest
        a2.computeInterest(); // → Savings account: 3% interest
    }
}
```

Without polymorphism, you'd write:

```
public void computeInterest(Account acc) {
    if (acc instanceof CheckingAccount) {
        System.out.println("Checking account: no interest");
    } else if (acc instanceof SavingsAccount) {
        System.out.println("Savings account: 3% interest");
    }
}
```

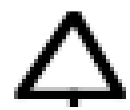
2.5 Organizing Classes into Inheritance Hierarchies

Superclasses

- Contain features common to a set of subclasses

Inheritance hierarchies

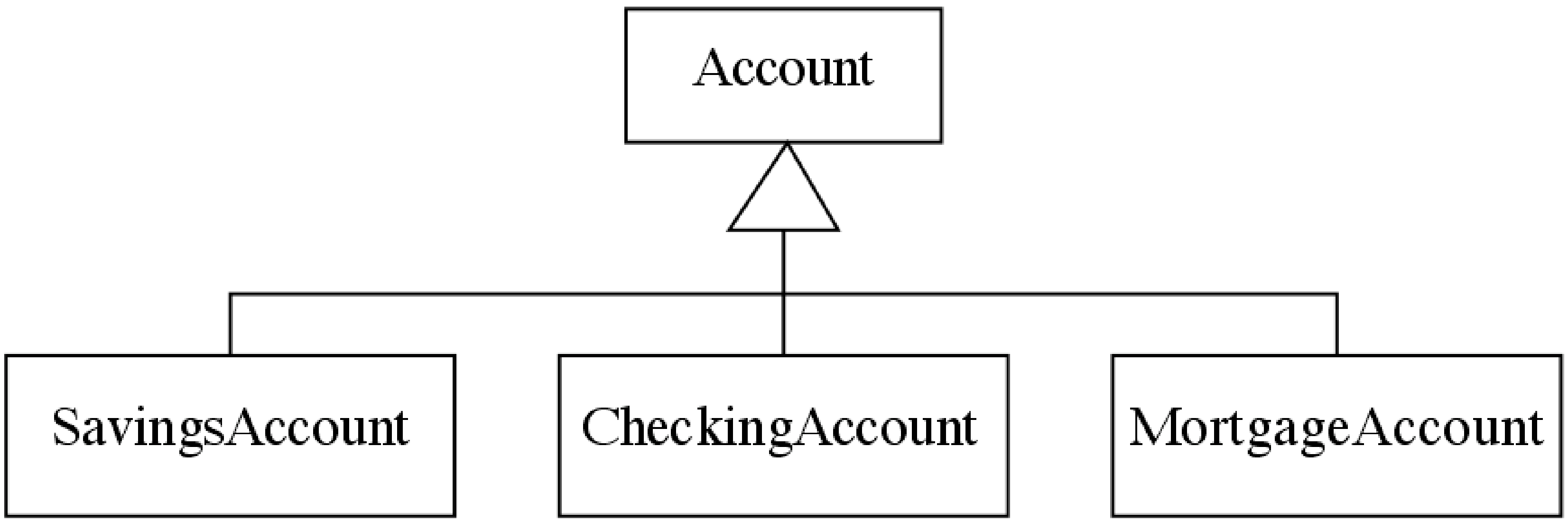
- Show the relationships among superclasses and subclasses
- A triangle shows a *generalization*



Inheritance

- The *implicit* possession by all subclasses of features defined in its superclasses

An Example Inheritance Hierarchy



Inheritance

- The *implicit* possession by all subclasses of features defined in its superclasses

The Isa Rule

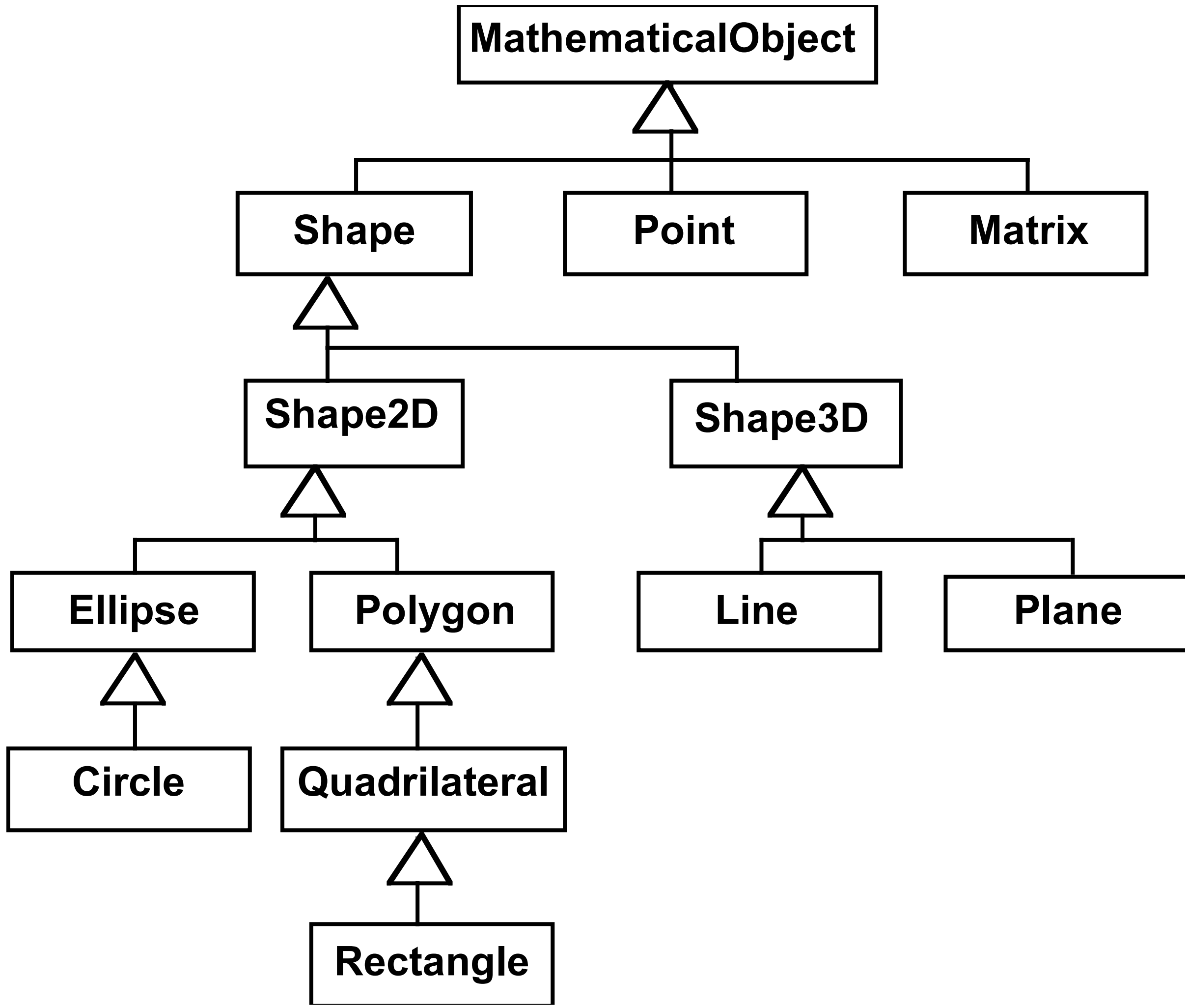
Always check generalizations to ensure they obey the isa rule

- "A checking account ***is an*** account"
- "A village ***is a*** municipality"

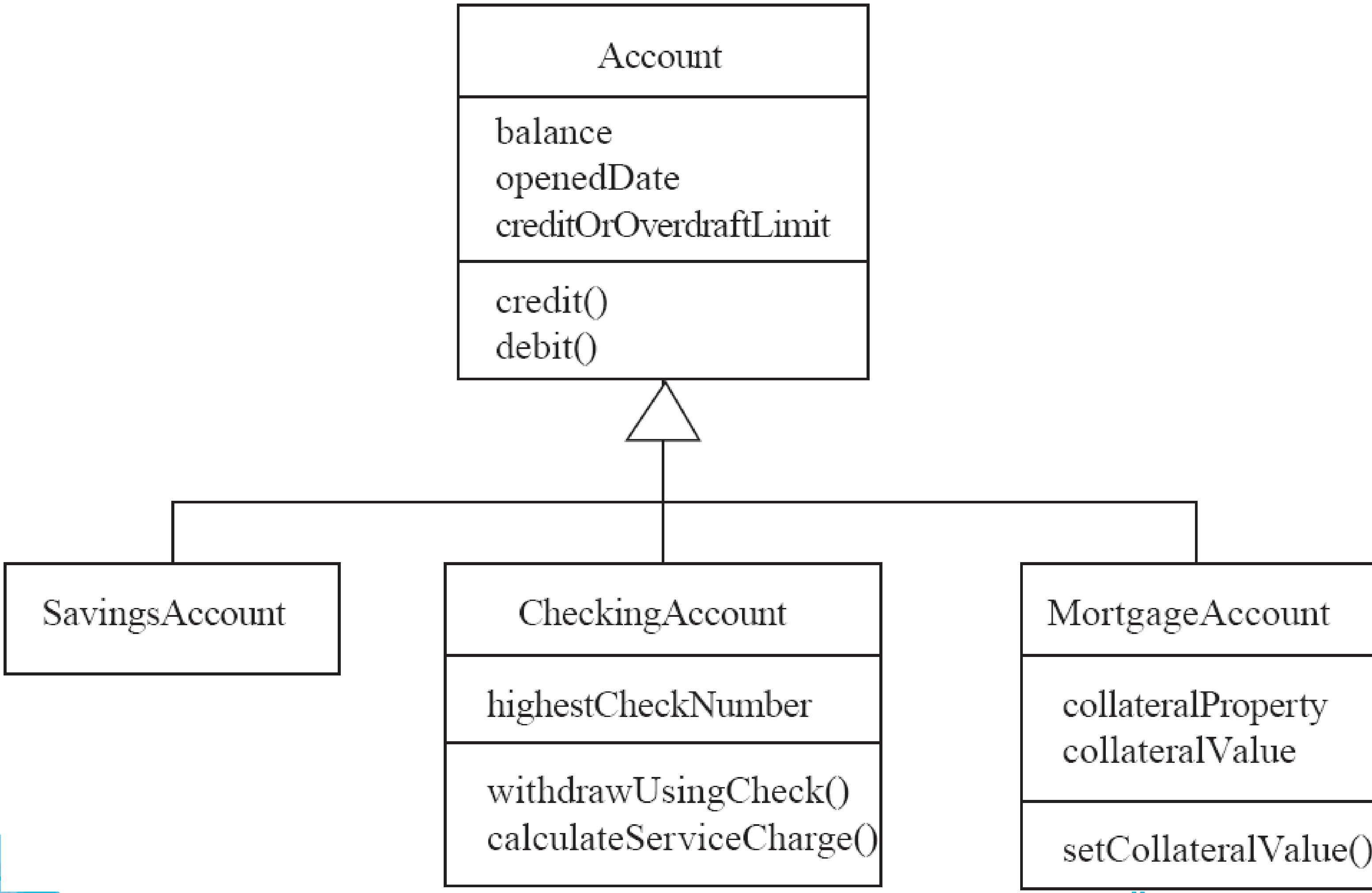
Should 'Province' be a subclass of 'Country'?

- No, it violates the isa rule
  - "A province ***is a*** country" is invalid!

A possible inheritance hierarchy of mathematical objects



Make Sure all Inherited Features Make Sense in Subclasses



when designing subclasses, you should only inherit features that are logically applicable.

Imagine the `creditOrOverdraftLimit` attribute in the Account class:

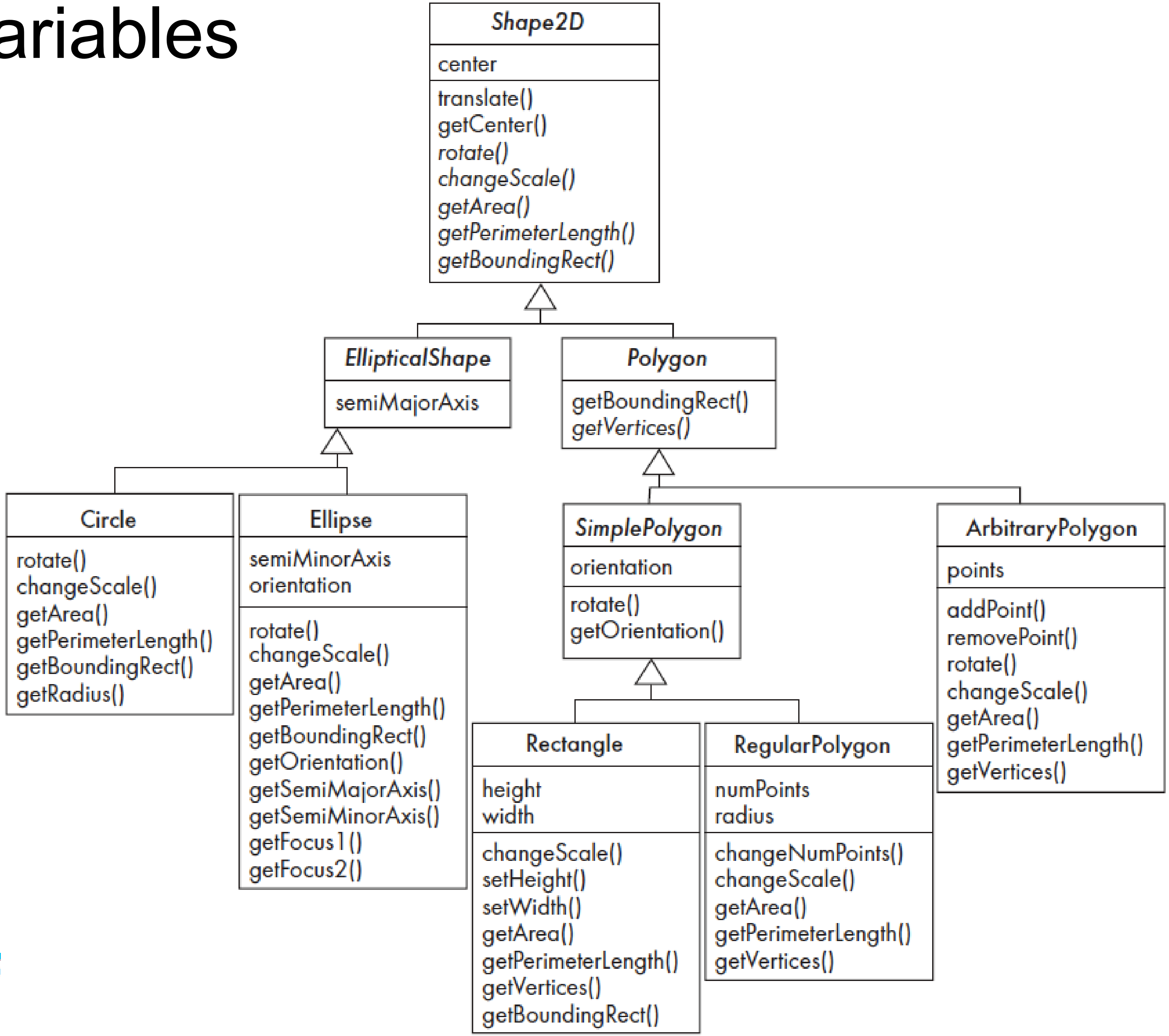
It makes sense for a `CheckingAccount` (which might allow overdrafts).

But it might not make sense for a `MortgageAccount`, which is typically a loan and doesn't have an overdraft feature.

So, if `MortgageAccount` inherits `creditOrOverdraftLimit`, the design might be flawed unless it's justified.

```
abstract class Account {
    double balance;
    Date openedDate;

    abstract void credit(double amount);
    abstract void debit(double amount);
}

class CheckingAccount extends Account {
    double creditOrOverdraftLimit;
    int highestCheckNumber;

    void withdrawUsingCheck() { /*...*/ }
    void calculateServiceCharge() { /*...*/ }
}

class MortgageAccount extends Account {
    String collateralProperty;
    double collateralValue;

    void setCollateralValue(double value) { /*...*/ }
}
```

2.6 Inheritance, Polymorphism and Variables



Some Operations in the Shape Example

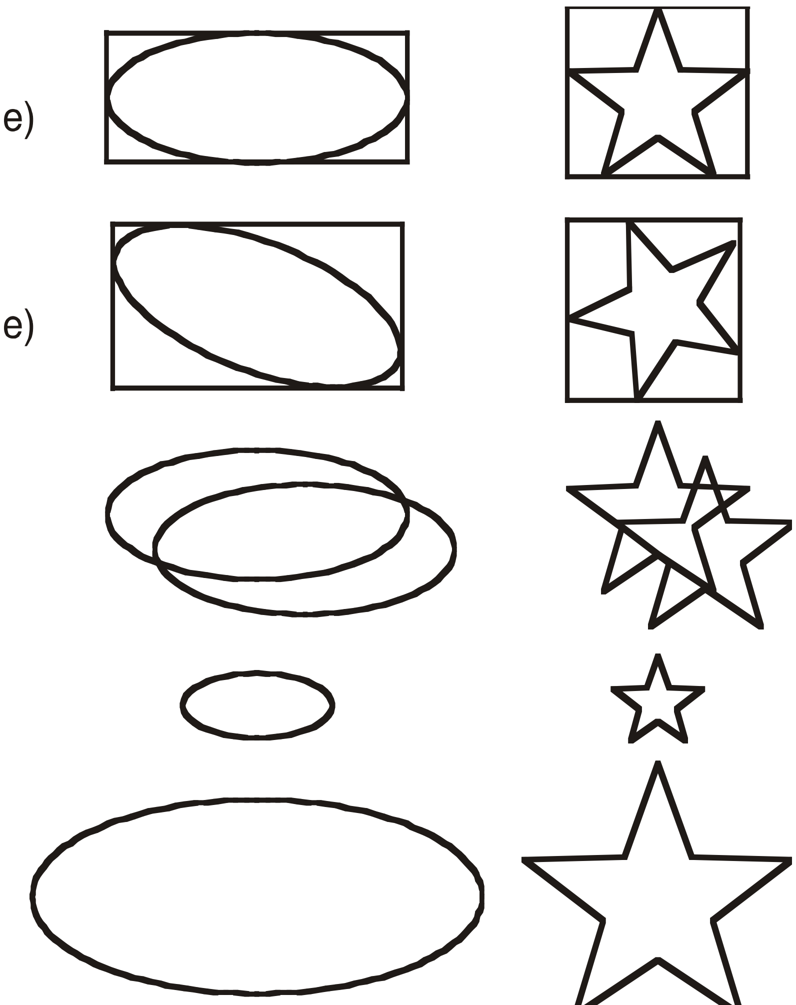
Original objects
(showing bounding rectangle)

Rotated objects
(showing bounding rectangle)

Translated objects
(showing original)

Scaled objects
(50%)

Scaled objects
(150%)



Abstract Classes and Methods

An operation should be declared to exist at the highest class in the hierarchy where it makes sense

- The *operation* may be *abstract* (lacking implementation) at that level
- If so, the *class* also <u>must</u> be *abstract*
  - —No instances can be created
  - —The opposite of an abstract class is a *concrete* class
- If a superclass has an abstract operation then its subclasses at some level must have a concrete method for the operation
  - —Leaf classes must have or inherit concrete methods for all operations
  - —Leaf classes must be concrete

Overriding

When a subclass provides its own version of a method that is already defined in its superclass, it's called method overriding.

There are **three common reasons** to override a method:

- For extension
  - —E.g. `SavingsAccount` might charge an extra fee following every debit
- For optimization
  - —E.g. The `getPerimeterLength` method in `Circle` is much simpler than the one in `Ellipse`
- For restriction (best to avoid)
  - —E.g. `scale(x,y)` would not work in `Circle`

```java
class SavingsAccount extends Account {
    @Override
    void debit(double amount) {
        super.debit(amount); // call base method
        chargeFee();         // extra behavior
    }
}
```

```java
class Circle extends Ellipse {
    @Override
    double getPerimeterLength() {
        return 2 * Math.PI * radius; // much simpler
    }
}
```

```java
class Circle extends Shape {
    @Override
    void scale(double x, double y) {
        if (x != y) throw new IllegalArgumentException("scale uniformly");
        super.scale(x, y);
    }
}
```

How a decision is made about which method to run

1. **If there is a concrete method for the operation in the current class, run that method.**
2. **Otherwise, check in the immediate superclass to see if there is a method there; if so, run it.**
3. **Repeat step 2, looking in successively higher superclasses until a concrete method is found and run.**
4. **If no method is found, then there is an error**
   - In Java and C++ the program would not have compiled

```java
class Shape {
    // No describe() method here
}

class Rectangle extends Shape {
    void describe() {
        System.out.println("I am a Rectangle.");
    }
}

class Square extends Rectangle {
    // No describe() method here
}

public class Main {
    public static void main(String[] args) {
        Square sq = new Square();
        sq.describe(); // What happens here?
    }
}
```

```java
Square sq = new Square();
sq.describe(); // ❌ Compile-time error: method not found

```

Dynamic binding

Occurs when decision about which method to run can only be made at *run time*

- Needed when:
  - —A variable is declared to have a superclass as its type, and
  - —There is more than one possible polymorphic method that could be run among the type of the variable and its subclasses

We have a superclass Animal and two subclasses: Dog and Cat. Each subclass overrides the speak() method.

```
class Animal {
    void speak() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {
    @Override
    void speak() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {
    @Override
    void speak() {
        System.out.println("Meow!");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Animal a1 = new Dog();  // declared as Animal, actual type is Dog
        Animal a2 = new Cat();  // declared as Animal, actual type is Cat

        a1.speak();  // Output: Woof!
        a2.speak();  // Output: Meow!
    }
}
```

a1 and a2 are declared as type Animal.

At **compile time**, Java only knows they are Animal.

At **runtime**, Java checks the **actual object type** (Dog or Cat) and runs the correct speak() method.

This is **dynamic binding** (also called **late binding**).

Key Terminology

Abstraction

- Object -> something in the world
- Class -> objects
- Superclass -> subclasses
- Operation -> methods
- Attributes and associations -> instance variables

Modularity

- Code is divided into classes, and classes into methods

Encapsulation

- Details can be hidden in classes
- This gives rise to *information hiding*:
    —Programmers do not need to know all the details of a class

The Basics of Java

History

- The first object oriented programming language was Simula-67
  - —designed to allow programmers to write simulation programs
- In the early 1980's, Smalltalk was developed at Xerox PARC
  - —New syntax, large open-source library of reusable code, bytecode, platform independence, garbage collection.
- late 1980's, C++ was developed by B. Stroustrup,
  - —Recognized the advantages of OO but also recognized that there were tremendous numbers of C programmers
- In 1991, engineers at Sun Microsystems started a project to design a language that could be used in consumer 'smart devices': Oak
  - —When the Internet gained popularity, Sun saw an opportunity to exploit the technology.
  - —The new language, renamed Java, was formally presented in 1995 at the SunWorld '95 conference.

Java documentation

Looking up classes and methods is an essential skill

- Looking up unknown classes and methods will get you a long way towards understanding code

Java documentation can be automatically generated by a program called Javadoc

- Documentation is generated from the code and its comments
- You should format your comments as shown in some of the book's examples
  - —These may include embeded html

Overview of Java

The next few slides will remind you of several key Java features

- Not in the book
- See the book's web site for
  - —A more detailed overview of Java
  - —Pointers to tutorials, books etc.

Characters and Strings

`Character` **is a class representing Unicode characters**

- More than a byte each
- Represent any world language

`char` **is a primitive data type containing a Unicode character**

`String` **is a class containing collections of characters**

- + is the operator used to concatenate strings

Arrays and Collections

Arrays are of fixed size and lack methods to manipulate them

`ArrayList` is the most widely used class to hold a *collection* of other objects

- More powerful than arrays, but less efficient

`Iterators` are used to access members of `Vectors`

- Enumerations were formally used, but were more complex

```
a = new ArrayList();
Iterator i = a.iterator();
while(i.hasNext())
{
   aMethod(i.next());
}
```

Casting

Java is very strict about types

In Java, if a variable is declared as a **superclass**, you can only access methods defined in that superclass even if the actual object is a subclass.

Without Casting

```
class Animal {
    void speak() {
        System.out.println("Animal speaks");
    }
}

class Dog extends Animal {
    void fetch() {
        System.out.println("Dog fetches");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();  // a is declared as Animal, but actually a Dog
        a.speak();             // OK: speak() is in Animal
        // a.fetch();          // ❌ Compile-time error: fetch() not in Animal
    }
}
```

Wit Casting

```
class Animal {
    void speak() {
        System.out.println("Animal speaks");
    }
}

class Dog extends Animal {
    void fetch() {
        System.out.println("Dog fetches");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();  // a is declared as Animal, but actually a Dog
        a.speak();             // OK: speak() is in Animal
        Dog d = (Dog) a;   // Cast Animal to Dog
         d.fetch();         // ✅ Now you can call fetch()

    }
}
```

Exceptions

Anything that can go wrong should result in the raising of an Exception

- **Exception** is a class with many subclasses for specific things that can go wrong

Use a try - catch block to trap an exception

```
try
{
  // some code
}
catch (ArithmeticException e)
{
  // code to handle division by zero
}
```

Interfaces

Like abstract classes, but cannot have executable statements

- Define a set of operations that make sense in several classes
- Abstract Data Types

A class can implement any number of interfaces

- It must have concrete methods for the operations

You can declare the type of a variable to be an interface

- This is just like declaring the type to be an abstract class

Important interfaces in Java's library include

- Runnable, Collection, Iterator, Comparable, Cloneable

Packages and importing

A package combines related classes into subsystems

- All the classes in a particular directory

Classes in different packages can have the same name

- Although not recommended

***Importing* a package is done as follows:**

```
import finance.banking.accounts.*;
```

Access control

Applies to methods and variables

- `public`
  - —Any class can access
- `protected`
  - —Only code in the package, or subclasses can access
- (blank)
  - —Only code in the package can access
- `private`
  - —Only code written in the class can access
  - —Inheritance still occurs!

Threads and concurrency

Thread:

- Sequence of executing statements that can be running concurrently with other threads

To create a thread in Java:

- 1. Create a class implementing `Runnable` or extending `Thread`
- 2. Implement the `run` method as a loop that does something for a period of time
- 3. Create an instance of this class
- 4. Invoke the `start` operation, which calls `run`

Programming Style Guidelines

Remember that programs are for people to read

- Always choose the simpler alternative
- Reject clever code that is hard to understand
- Shorter code is not necessarily better

Choose good names

- Make them highly descriptive
- Do not worry about using long names

Programming style ...

Comment extensively

- Comment whatever is non-obvious
- Do not comment the obvious
- Comments should be 25-50% of the code

Organize class elements consistently

- Variables, constructors, public methods then private methods

Be consistent regarding layout of code

Programming style ...

Avoid duplication of code

- Do not 'clone' if possible
  - —Create a new method and call it
  - —Cloning results in two copies that may both have bugs
    - When one copy of the bug is fixed, the other may be forgotten

Programming style ...

Adhere to good object oriented principles

- E.g. the 'isa rule'

Prefer `private` as opposed to `public`

Do not mix user interface code with non-user interface code

- Interact with the user in separate classes
  - —This makes non-UI classes more reusable

2.10 Difficulties and Risks in Programming

Language evolution and deprecated features:

- Java is evolving, so some features are 'deprecated' at every release

Efficiency can be a concern in some object oriented systems

- Java can be less efficient than other languages
  - —VM-based
  - —Dynamic binding