



CH. 2 OBJECT-ORIENTED PROGRAMMING

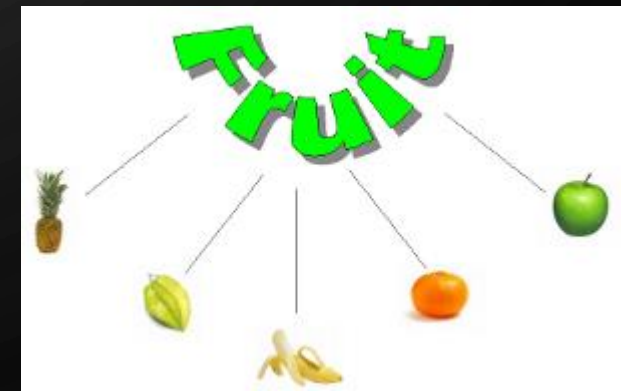
ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)

The image features a dark gray background with white, stylized circuit board traces in the corners. These traces consist of straight lines of varying lengths and angles, ending in small white circles, resembling a PCB layout. The patterns are located in the top-left, top-right, bottom-left, and bottom-right corners, framing the central text.

CRASH COURSE IN OBJECT-ORIENTED PROGRAMMING

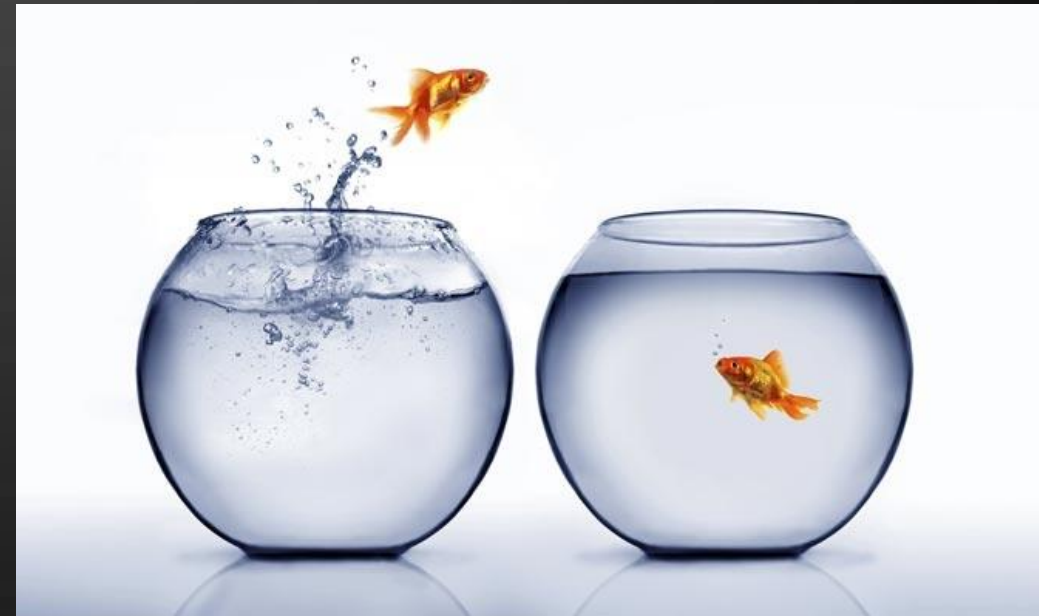
OBJECT-ORIENTED DESIGN PRINCIPLES

- **Object Oriented Programming** – paradigm for programming involving modularizing code into self contained **objects** that are a concise and consistent view of a “thing” without exposing unnecessary detail like the inner workings of the object
 - Composition/Abstraction – What makes up an object? The model
 - Encapsulation – Hiding implementation details, only exposing the “public interface”
 - Inheritance – Types and subtypes, it’s a modeling decision
 - Polymorphism – Provision of a single interface to entities of different types




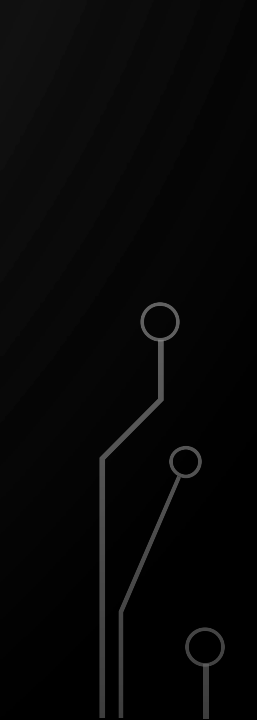
GOALS

- **Robustness**
 - We want software to be capable of handling unexpected inputs that are not explicitly defined for its application.
- **Adaptability**
 - Software needs to be able to evolve over time in response to changing conditions in its environment.
- **Reusability**
 - The same code should be usable as a component of different systems in various applications.





OBJECT-ORIENTED SOFTWARE DESIGN

- Responsibilities
 - Divide the work into different actors, each with a different responsibility.
 - Independence
 - Define the work for each class to be as independent from other classes as possible.
 - Behaviors
 - Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.
- 
- 

TERMINOLOGY

- “object” is a little ambiguous
 - Object type or **class** – specifies **instance variables**, also known as **data members**, that the object contains, as well as the **methods**, also known as **member functions**, that the object can execute
 - Object **instance**, i.e., variable of that object type

CLASS DEFINITIONS

- A class serves as the primary means for abstraction in object-oriented programming.
- In Java, every variable is either a primitive type or is a reference to an instance of some class
- A class provides a set of behaviors in the form of member functions (also known as methods), with implementations that belong to all its instances.
- A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of attributes (also known as fields, instance variables, or data members).

USING A CLASS (QUICK AND DIRTY REFRESHER)

- Initialize a variable of an object with the keyword **new** followed by a call to a **constructor** of the object:

```
String s = new String("Hello");
```

- Use a method of the class to execute a computation:

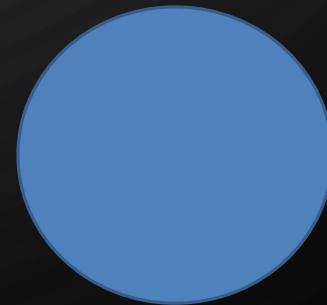
```
int l = s.length();
```


CLASS TEMPLATE (QUICK AND DIRTY REFRESHER)

```
1. public class ClassName {
2.     /* All instance variables declared private*/
3.     private int i = 0;
4.     /* Any public static final variables - these model constants */
5.     /* All constructors - constructors initialize all member data,
        and must be named the same as the class */
6.     public ClassName() {}
7.     /* All accessor (getters) and simple modifiers (setters) needed
        for the object */
8.     public int getI() {return i;}
9.     public int setI(int i) {this.i = i;}
10.    /* All other public methods */
11.    /* Any and all private methods */
12.    /* Any and all static methods */
13.}
```

EXAMPLE

- Lets program (in pairs) a class for a circle, Circle.java
 - Have getter and setter for private member data
 - Have an area computation
 - Have a computation to determine if two circles overlap
- Program a simple test to exercise all of the methods of circle



ABSTRACT DATA TYPES

- **Abstraction** is to distill a system to its most fundamental parts.
- Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs).
- An ADT is a model of a data structure that specifies the **type** of data stored, the **operations** supported on them, and the types of parameters of the operations.
 - This would essentially be the “public interface” of a class
- ***An ADT specifies what each operation does, but not how it does it***
 - Lets repeat, an ADT is the operations not the implementation!
 - We will see that we can implement ADTs in many, many ways

INTERFACES AND ABSTRACT CLASSES

- The main structural element in Java that enforces an application programming interface (API) is an interface.
- An **interface** is a collection of method declarations with no data and no bodies.
- Interfaces do not have constructors and they cannot be directly instantiated.
 - When a class **implements** an interface, it must implement all of the methods declared in the interface.
- An **abstract class** also cannot be instantiated, but it can define one or more common methods that all implementations of the abstraction will have.

INTERFACE EXAMPLE

```
1. public interface Robot {  
2.     void sense(World w);  
3.     void plan();  
4.     void act(World w);  
5. }
```



USE IMPLEMENTS TO ENFORCE THE INTERFACE

```
1. public class Roomba implements Robot {  
2.     /* code specific to Roomba */  
3.     public void sense(World w) { /* Roomba's don't sense */}  
4.     public void plan() { /* code for roombas actions */}  
5.     public void act(World w) { /* code to power motors */}  
6.     /* code specific to Roomba */  
7. }
```

INHERITANCE

- A mechanism for a modular and hierarchical organization is inheritance.
- This allows a new class to be defined based upon an existing class as the starting point.
- The existing class is typically described as the base class, parent class, or superclass, while the newly defined class is known as the subclass or child class.
- There are two ways in which a subclass can differentiate itself from its superclass:
 - A subclass may specialize an existing behavior by providing a new implementation that overrides an existing method.
 - A subclass may also extend its superclass by providing brand new methods.
- Used to support run-time polymorphism


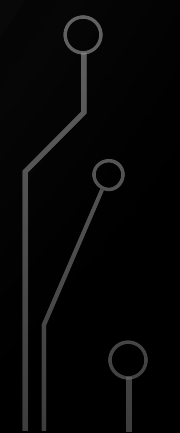
The image features a dark gray background with white, stylized circuit board traces in the corners. These traces consist of straight lines of varying lengths and angles, ending in small white circles, resembling a PCB layout. The patterns are located in the top-left, top-right, bottom-left, and bottom-right corners, framing the central text.

ADVANCED PROGRAMMING TECHNIQUES



EXCEPTIONS



- **Exceptions** are unexpected events that occur during the execution of a program.
 - An exception might result due to an unavailable resource, unexpected input from a user, or simply a logical error on the part of the programmer.
 - In Java, exceptions are objects that can be **thrown** by code that encounters an unexpected situation.
 - An exception may also be **caught** by a surrounding block of code that “handles” the problem.
 - If uncaught, an exception causes the virtual machine to stop executing the program and to report an appropriate message to the console.
- 
- 

CATCHING EXCEPTIONS

- The general methodology for handling exceptions is a **try-catch** construct in which a guarded fragment of code that might throw an exception is executed.
- If it **throws** an exception, then that exception is caught by having the flow of control jump to a predefined **catch block** that contains the code to apply an appropriate resolution.
- If no exception occurs in the guarded code, all catch blocks are ignored.

```
1 . ...
2 . try {
3 .     /*Code that may
         generate exception*/
4 . }
5 . catch (ExceptionType1 e1) {
6 . }
7 . catch (ExceptionType2 e2) {
8 . }
9 . ...
```

THROWING EXCEPTIONS

- Exceptions originate when a piece of Java code finds some sort of problem during execution and throws an exception object.
- This is done by using the **throw** keyword followed by an instance of the exception type to be thrown:

```
throw new ExceptionType (parameters) ;
```

THE THROWS CLAUSE

- When a method is declared, it is possible to explicitly declare, as part of its signature, the possibility that a particular exception type may be thrown during a call to that method.
- The syntax for declaring possible exceptions in a method signature relies on the keyword **throws** (not to be confused with an actual throw statement).

```
public static int parseInt(String s)  
    throws NumberFormatException;
```

CASTING

- Casting with Objects allows for conversion between classes and subclasses.
- A **widening conversion** occurs when a type T is converted into a “wider” type U:
 - T is a type and U is a superclass/superinterface of T
- Example: **Shape** s = **new Circle** ();

CASTING

- A **narrowing conversion** occurs when a type T is converted into a “narrower” type S.
 - S is a type and T is a superclass/superinterface of S
- A narrowing conversion of reference types requires an explicit cast.
- Example: **Circle** c = (**Circle**) s;

GENERIC PROGRAMMING

- **Generic Programming** is a programming paradigm where the programmer programs interfaces and algorithms without a specific object hierarchy in mind. Rather the programmer only worries about operations (methods) needed to support an interface or perform an algorithm
- Java includes support for writing generic classes and methods, called **Generics**
- The actual type parameters are later specified when using the generic class/method as a type elsewhere in a program. Determined at compile-time not run-time!

SYNTAX FOR GENERICS

- Types can be declared using generic names:

```
1. public class Node<E> {  
2.     private E e;  
3.     private Node<E> next;  
4.     /* Rest of linked structure */  
5. }
```

- They are then instantiated using actual types:

- `Node<String> head = new Node<>();`


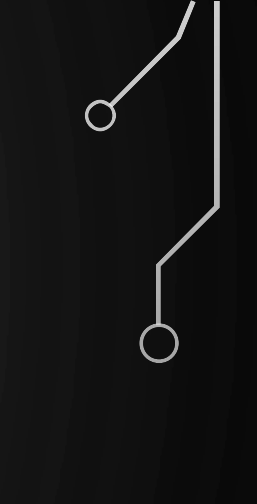
- There is not much to it actually, but it is a very strange thought process that you do not know what E is as you write it.

NESTED CLASSES

- Java allows a class definition to be nested inside the definition of another class.
- The main use for nesting classes is when defining a class that is strongly affiliated with another class.
 - This can help increase encapsulation and reduce undesired name conflicts.
- Nested classes are a valuable technique when implementing data structures, as an instance of a nested use can be used to represent a small portion of a larger data structure, or an auxiliary class that helps navigate a primary data structure.



EXERCISE

- With a team, program a generic singly-linked list class with a generic nested node class that supports
 - `integer size()`
 - `boolean isEmpty()`
 - `addFirst(e)`
 - `removeFirst(e)`
- 
- 
- 