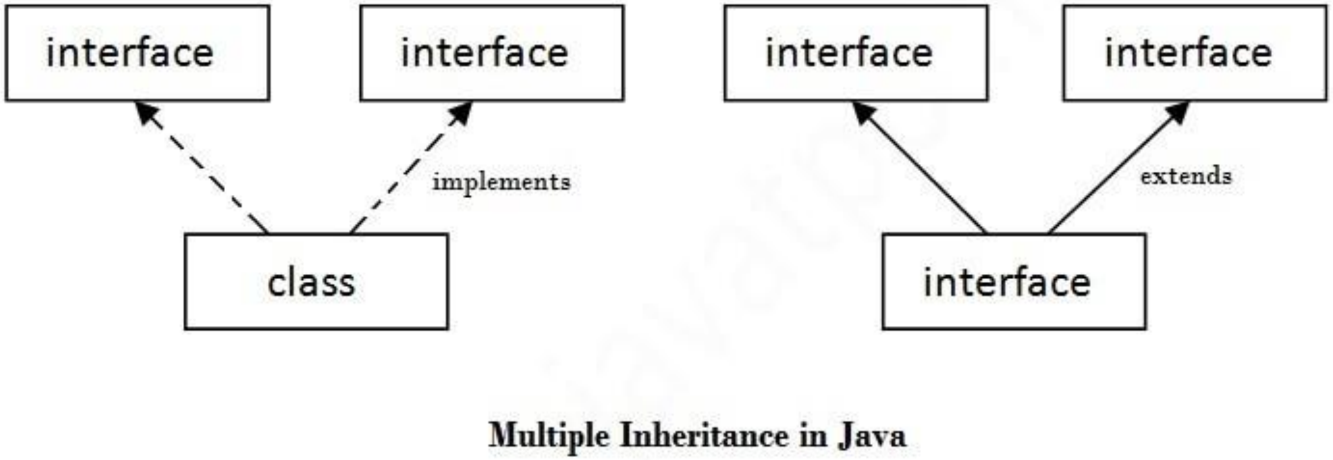
Java Interfaces



Computer Science OOD
Boston University

Christine Papadakis-Kanaris

Rectangles can be drawn!

```
public interface Drawable {
    void draw();
}
```

Rectangles can be drawn!

```
public interface Drawable {
    void draw();
}

class Rectangle extends Shape
{
    public void draw() {
      ...
    }
}

class abstract Shape implements Drawable {

}
```

Recta

Method `draw()` is an *abstract* method of the `Drawable` interface, but as `Shape` is an abstract class, it passes that responsibility to the subclass!

```
public interface
    void draw()
}

class Rectangle extend  Shape
{
    public void draw() { // must be implemented
      ...
    }
}

class abstract Shape implements Drawable {

}
```

Draw, an abstract method of Drawable interface

```
public class DisplayShapes {

    public static void main( String[] s ) {

        Shape arr[] = { new Rectangle()
                      , new Circle()
                      , new Square() };


        for ( Shape s: arr )
            s.draw();
    }
}
```

Draw, an abstract method of *Drawable* interface

```
public class DisplayShapes {

    public static void main( String[] s ) {

        Shape arr[] = { new Rectangle()
                      , new Circle()
                      , new Square() };


        for ( Shape s: arr )
            s.draw();
    }
}
```

Draw, an abstract method of *Drawable* interface

```
public class DisplayShapes {

    public static void main( String[] s ) {

        Shape arr[] = { new Rectangle()
                      , new Circle()
                      , new Square() };


        for ( Shape s: arr )
            s.draw();
    }
}
```

Draw, an abstract method of *Drawable* interface

```
public class DisplayShapes {

    public static voi

        Shape arr[] =
                      , ne
                        w Square() };


        for ( Shape s: arr )
            s.draw();
    }
}
```

Can call methods on `s` that are known to the Shape class (i.e. objects of type Shape).

Draw, an abstract method of *Drawable* interface

Drawing objects of different types?

```
public class DisplayShapes {

    public static void main( String[] s ) {

        ???? arr[] = { new Rectangle()
                     , new Circle()
                     , new Cat()
                     , new Dog() };

        for ( ???? d: arr )
            d.draw();
    }
}
```

Draw, an abstract method of Drawable interface

Drawing objects of different types?

```
public class DisplayShapes {

    public static void main( String[] s ) {

        Drawable arr[] = { new Rectangle()
                         , new Circle()
                         , new Cat()
                         , new Dog() };

        for ( Drawable d: arr )
            d.draw();
    }
}
```

Draw, an abstract method of *Drawable* interface

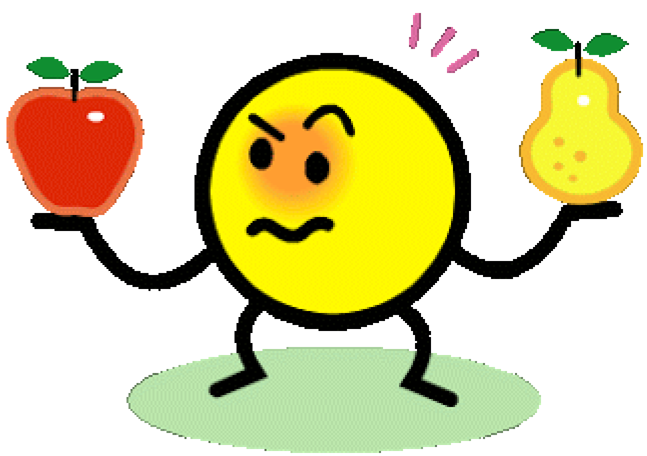
Drawing objects of different types?

```
public class DisplayShapes {

    public static voi

        Drawable arr[
                          , ne
                               w Cat()
                          , new Dog() };

        for ( Drawable d: arr )
            d.draw();
    }
}
```

Can only call methods on d that
are known to the Drawable type
(i.e. objects of type Drawable).

Comparing Objects in Java

Creating objects
that are *comparable*
in Java!



How to Compare **Objects**

- We need to be able to compare items in the heap.
- If those items are objects, we cannot use relational operators:

```
    if (item1 < item2)
```

Why not?

How to Compare Objects

- We need to be able to compare items in the heap.

- If those items are objects, we cannot use relational operators:

```
if (item1 < item2)
```

  Why not?
  this compares the references, not the objects' fields

- Instead, (in Java) we need to use a method to compare them.

compareTo

How to Compare Objects

- We need to be able to compare items in the heap.

- If those items are objects, we cannot use relational operators:

```
if (item1 < item2)
```

  Why not?
  this compares the references, not the objects' fields

- Instead, (in Java) we need to use a method to compare them.

compareTo

Implement the..

Comparable Interface

…to ensure that our class can use this method.

An Interface for Objects That Can Be Compared

```
public interface Comparable {
    public int compareTo(Object other);
}
```

- `item1.compareTo(item2)` should return:
  - a negative integer if `item1` "comes before" `item2`
  - a positive integer if `item1` "comes after" `item2`
  - 0 if `item1` and `item2` are equivalent in the ordering
- These conventions make it easy to construct appropriate method calls:

numeric comparison	comparison using `compareTo`
`item1 < item2`	`item1.compareTo(item2) < 0`
`item1 > item2`	`item1.compareTo(item2) > 0`
`item1 == item2`	`item1.compareTo(item2) == 0`

An Interface for Objects That Can Be Compared

- The `Comparable` interface is a built-in generic Java interface:

```
public interface Comparable {
    public int compareTo(Object other);
}
```

- It is used when defining a class of objects that can be ordered.

- Examples from the built-in Java classes:

```
public class String implements Comparable<String> {
    ...
    public int compareTo(String other) {
        ...
}

public class Integer implements Comparable<Integer> {
    ...
    public int compareTo(Integer other) {
        ...
}
```

Comparable Objects

```
public class Max {
    public static Object max( Object o1, Object o2 ) {

        if ( o1.compareTo(o2) > 0 )
            return( o1 );
        else
            return(o2);
    }
}
```

Comparable Objects

```
public class Max {
    public static Object max( Object o1, Object o2 ) {

        if ( o1.compareTo(o2) > 0 )
            return( o1 );
        else
            return(o2);
    }
}
```

Cannot invoke the `compareTo` method on an object of type `Object` because the Object class does not implement the `Comparable` interface!

Comparable Objects

```
public class Max {
    public static Object max( Object o1, Object o2 ) {

        if ( ((Comparable) o1).compareTo(o2) > 0 )
            return( o1 );
        else
            return(o2);
    }
}
```

can cast the
reference or …

Comparable Objects

```
public class Max {
    public static Object max( Object o1, Object o2 ) {

        if ( ((Comparable) o1).compareTo(o2) > 0 )
            return( o1 );
        else
            return(o2);
    }
}
```

Can have a potential run-time error if the object passed is not Comparable!

Comparable Objects

```
public class Max {
    public static Object max( Comparable o1, Comparable o2 ) {

        if ( o1.compareTo(o2) > 0 )
            return( o1 );
        else
            return(o2);
    }
}
```

… type the method to only accept `Compareable` objects!

Comparable Objects

```
public class Max {
    public static Object max( Comparable o1, Comparable o2 ) {

        if ( o1.compareTo(o2) > 0 )
            return( o1 );
        else
            return(o2);
    }
}
```

Objects of classes that implement the `Comparable` interface are themselves Comparable!

Operator Overloading in OO

C++

```
class className {

private:

public:
...
    // Operator overloaded
    operator==(ClassName) {..}
    operator<(ClassName) {..}
    operator<=(ClassName) {..}
...
}
```

Python

```
class className {
...
    // Operator overloaded
    __eq__(ClassNam) { .. }
    __add__(ClassName) { .. }
    __mul__(ClassName) { .. }
...
}
```

Operator Overloading in OO

C++

```
class className {

private:

public:
...
    // Operator overloaded
    operator==(ClassName) {..}
    operator<(ClassName) {..}
    operator<=(ClassName) {..}
...
}
```

```
{
    ClassName o1, o2;

    o1 == o2;
    o1 <= o2;
}
```

Python

```
class className {
...
    // Operator overloaded
    __eq__(ClassNam) { .. }
    __add__(ClassName) { .. }
    __mul__(ClassName) { .. }
...
}
```

```
{
    ClassName o1 = new ...
    ClassName o2 = new ...

    o1 == o2
    o1 <= o2
    o1 + o2
    o1 <= o2
}
```

Operator Overloading in OO

C++

```
class className {

private:

public:
...
    // Operator overloaded
    operator==(Cl
    operator<(Cla
    operator<=(Cl
...
}
```

```
{
    ClassName o1, o2;

    o1 == o2;
    o1 <= o2;
}
```

Compiler expands to:

```
o1.operator==(o2);
o1.operator<=(o2);
```

Python

```
class className {
...
    // Operator overloaded
    __eq__(ClassNam) { .. }
    __add__(ClassName) { .. }
    __mul__(ClassName) { .. }
```

```
= new ...
= new ...

    o1 == o2
    o1 <= o2
    o1 + o2
    o1 <= o2
}
```

Operator Overloading in OO

C++

```
class className {

private:

public:
...
    // Operator overl
    operator==(ClassN
    operator<(ClassNa
    operator<=(ClassN
...
}
```

```
{
    ClassName o1, o2;

    o1 == o2;
    o1 <= o2;
}
```

Python

```
class className {
...
    // Operator overloaded
    __eq__(ClassNam) { .. }
    __add__(ClassName) { .. }
    __mul__(ClassName) { .. }
```

Interpreter expands to:

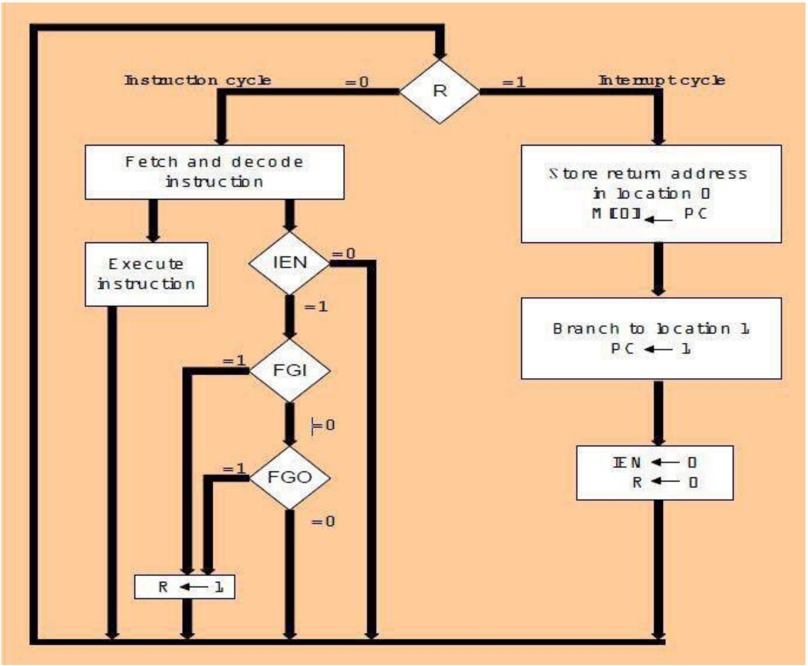
```
o1.__eq__(o2);
o1.__leq__(o2);
o1.__add__(o2);
o1.__geq__(o2);
```

```
ew ...
ew ...

    o1 == o2
    o1 <= o2
    o1 + o2
    o1 >= o2
}
```

Interfaces vs. Abstract Classes

Classes provide a means to maintain State

Contract



Interfaces vs. Abstract Classes

	Variables	Constructors	Methods
Abstract Classes	No restrictions.	Constructors are invoked by subclasses through constructor chaining.	No restrictions.
Interfaces	All variables are public static final	No constructors.	All methods must be public abstract instance methods.

Interfaces vs. Abstract Classes

	Variables	Constructors	Methods
Abstract Classes	No restrictions.	Constructors are invoked by subclasses through constructor chaining.	No restrictions.
Interfaces	All variables are public static final	No constructors.	All methods must be public abstract instance methods.

Interfaces vs. Abstract Classes

	Variables	Constructors	Methods
Abstract Classes	No restrictions.	Constructors are invoked by subclasses through constructor chaining.	No restrictions.
Interfaces	All variables are public static final	No constructors.	All methods must be public abstract instance methods.

Interfaces vs. Abstract Classes

	Variables	Constructors	Methods
Abstract Classes	No restrictions.	Constructors are invoked by subclasses through constructor chaining.	No restrictions.
Interfaces	All variables are public static final	No constructors.	All methods must be public abstract instance methods.

Interfaces vs. Abstract Classes

	Variables	Constructors	Methods
Abstract Classes	No restrictions.	Constructors are invoked by subclasses through constructor chaining.	No restrictions.
Interfaces	All variables are public static final	No constructors.	All methods must be public abstract instance methods.

The only variables declared in the interface are those that will have class scope and are read only!

Interfaces vs. Abstract Classes

	Variables	Constructors	Methods
Abstract Classes	No restrictions.	Constructors are invoked by subclasses through …	No restrictions.
Interfaces	All variables are public static final	No constructors.	All methods must be public abstract instance methods.

Interfaces vs. Abstract Classes

	Variables	**Constructors**	**Methods**
Abstract Classes	No restrictions.	Constructors are invoked by subclasses through constructor chaining.	No restrictions.
Interfaces	All variables are public static final	No constructors.	All methods must be public abstract instance methods.

Cannot create objects of abstract classes but the constructors are invoked when creating objects of the subclasses!

Interfaces vs. Abstract Classes

	Variables	Constructors	Methods
Abstract Classes	No restrictions.	Constructors are invoked by subclasses through constructor chaining.	No restrictions.
Interfaces	All variables are public static final	No constructors.	All methods must be public abstract instance methods.

Interfaces vs. Abstract Classes

	Variables	Constructors	Methods
Abstract Classes	No restrictions.	Constructors are invoked by subclasses through constructor chaining.	No restrictions.
Interfaces	All variables are public static final	No constructors.	All methods must be public abstract instance methods.

Cannot create objects from an interface!

Interfaces vs. Abstract Classes

	Variables	Constructors	Methods
Abstract Classes	No restrictions.	Constructors are invoked by subclasses through constructor chaining.	No restrictions.
Interfaces	All variables are public static final	No constructors.	All methods must be public abstract instance methods.

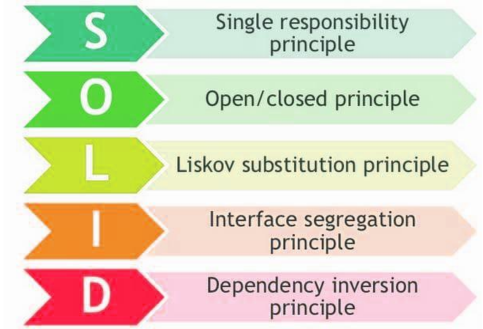
Interfaces vs. Abstract Classes

	Variables	Constructors	Methods
Abstract Classes	No restrictions.	Constructors are invoked by subclasses through constructor chaining.	No restrictions.
Interfaces	All variables are public static final	No constructors.	All methods must be public abstract *instance* methods.

Interfaces vs. Abstract Classes:
a summary

	Variables	Constructors	Methods
Abstract Classes	No restrictions.	Constructors are invoked by subclasses through constructor chaining.	No restrictions.
Interfaces	All variables are public static final	No constructors.	All methods must be public abstract *instance* methods.

Interface specifies the signature of each method.

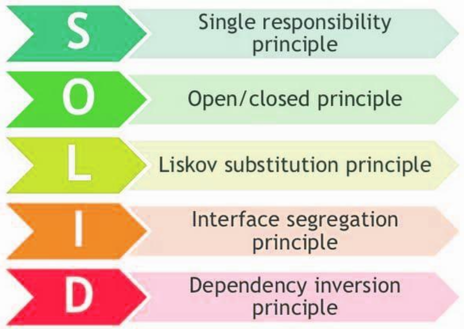


Decomposition *and* Abstraction

Single Responsibility Principle

limiting the impact of change

Open Closed Principle



Decomposition *and* Abstraction

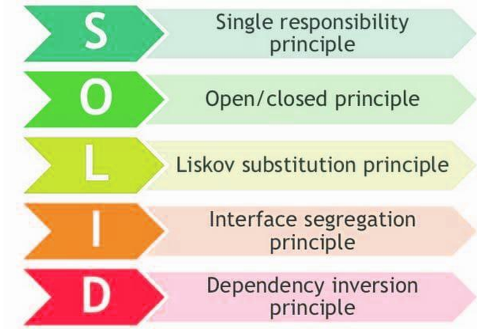
Single Responsibility Principle

a class should only **have one responsibility**, further defined by Martin as **'*one reason to change*'**

Robert Martin

limiting the impact of change

'***gather together those things that change for the same reasons***'



Decomposition *and* Abstraction

Open Close
Principle

A class should be **open** for *extension* but **closed** for *modification*.

limiting the impact of change

new features and behaviors should be able to be added to a class without requiring refactoring of existing code.

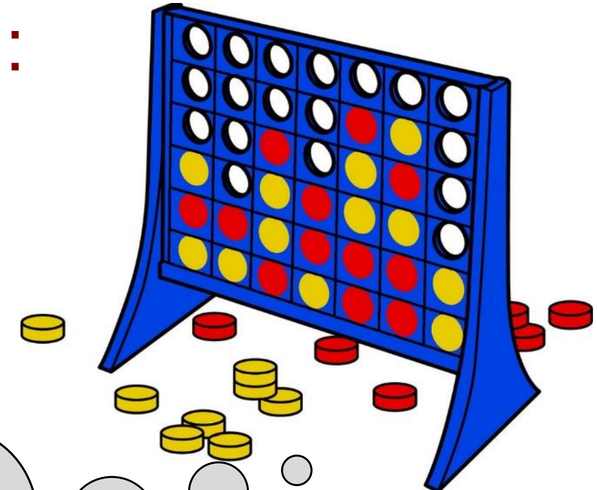
Principle of Abstraction

Purpose of abstraction is to handle the complexity of a software system by hiding unnecessary details from the user or client.

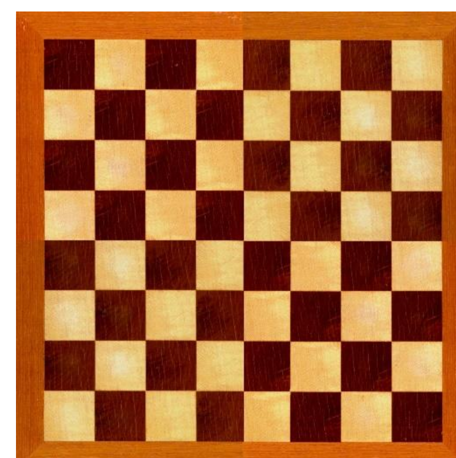
Abstraction assists us the process of ***decomposition***!

Object Decomposition:

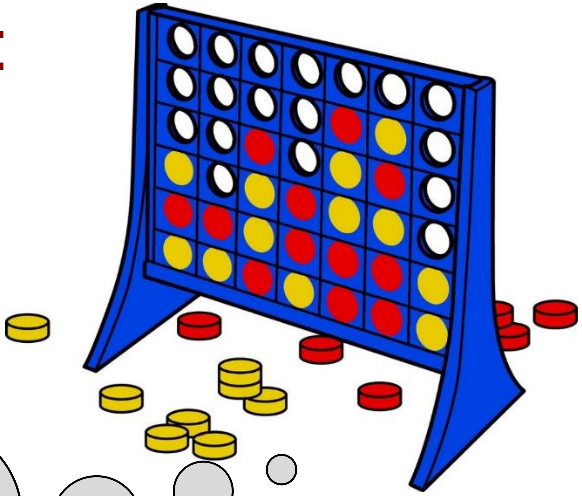
Principle of Abstraction

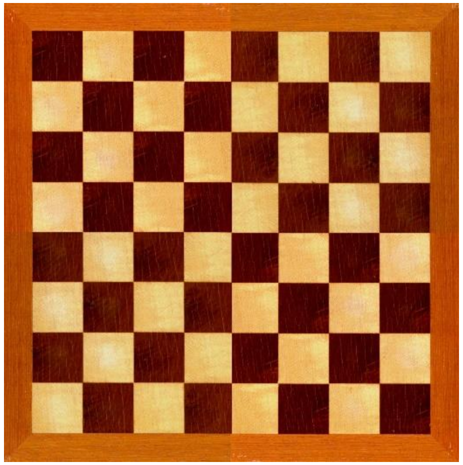


Four Individual Games



Object Decomposition:
Principle of Abstraction



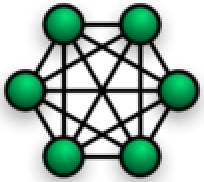


Quality of Abstraction

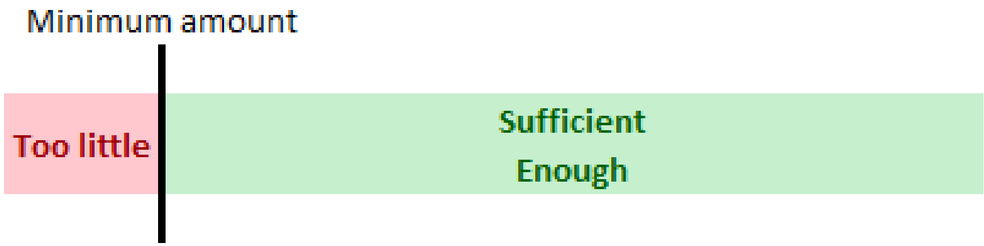
- How can determine if our class and object structure is well designed? Consider the following five factors.

1. Coupling

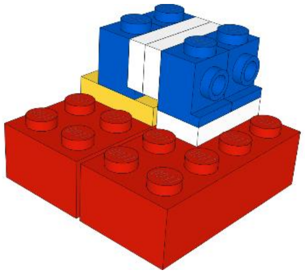
2. Cohesion



3. Sufficiency



4. Completeness
5. Primitiveness



Coupling

Strong

Implies a strong connection or dependencies between classes. We may not always want this. To maximize reuse classes should have a weak coupling so that they can be used independent of other classes.

Weak

Implies minimal if any dependencies between classes. Classes which are independent can be used as building blocks to form new programs.

Coupling

Strong

Implies a strong connection or dependencies between classes. We may not always want this. To maximize reuse class[illegible] should h[illegible] coupling so that they can be [illegible] other classes.

*However in OO Inheritance is one of the most powerful tools and this implies the **strongest** type of coupling!*

Weak

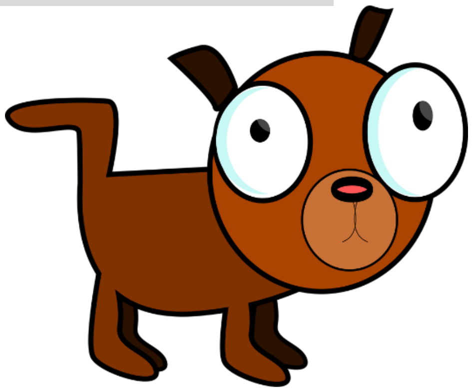
Implies minimal if any dependen[illegible]n classes. Classes which are independent can be used as building blocks to form new programs.

Cohesion

Measures the degree of relatedness among the elements (entities) of a single class.

All the members and methods of a class should work together to provide a clearly identified behavior of a specific entity.

Example, a class Dog is cohesive if its characteristics embrace the behavior of a dog and only a dog and not a cat who thinks she is a dog!



Sufficient, Complete and Primitive

Sufficient mean that the class captures enough characteristics of the abstraction to permit meaningful and efficient functionality of the concrete implementation. Complete means that the class captures ***all*** *the characteristics of the abstraction.*

Sufficient, Complete and Primitive

Sufficient mean that the class captures enough characteristics of the abstraction to permit meaningful and efficient functionality of the concrete implementation. Complete means that the class captures all the characteristics of the abstraction.

Let's say you are designing a class Set, we need to include operations that both add and remove items in the set. Neglecting one operation, does not allow us to meaningfully use it. Therefore, that class is not a sufficient implementation of a `Set`. However, if the class does not implement the difference operation, though it may not be complete, it can still be used.

Sufficient, Complete and Primitive

Sufficient mean that the class captures enough characteristics of the abstraction to permit meaningful and efficient functionality of the concrete implementation. Complete means that the class captures all the characteristics of the abstraction.

Let's say you are designing a class Set, we need to include operations that both add and remove items in the set. Neglecting one operation, does not allow us to meaningfully use it. Therefore, that class is not a sufficient implementation of a `Set`. However, if the class does not implement the difference operation, though it may not be complete, it can still be used.

Primitive means that the classes and objects should be designed as small independent building blocks which can be used to build higher level and more complex operations.

Mechanism of Abstraction

language based

- Abstraction by Parameterization
- Abstraction by Specification

Abstraction by Specification

The specification is a contract between the user and the developer.

It tells the client what can be relied upon when calling the function or method. The client should not assume anything about the behavior or implementation of the method. The specification also dictates to the developer of the method what behavior must be provided and the developer must meet the specification.

Abstraction by Specification

The specification is a contract between the client and the class.

It tells the client what can be relied upon when calling the function or method. The client should not assume anything about the behavior or implementation of the method. The specification also dictates to the developer of the method what behavior must be provided and the developer must meet the specification.

Abstraction by Specification

The specification is a contract between the client and the class.

It tells the client what can be relied upon when calling the function or method. The client should not assume anything about the behavior or implementation of the method. The specification also dictates to the developer of the method what behavior must be provided and the developer must meet the specification.

Abstraction by Specification

The specification is a contract between the client and the class.

It tells the client what can be relied upon when calling the function or method. The client should not assume anything about the behavior or implementation of the method. The specification also dictates to the developer of the method what behavior must be provided and the developer must meet the specification.

Abstraction by Specification

The specification is a contract between the client and the class.

It tells the client what can be relied upon when calling the function or method. The client should not assume anything about the behavior or implementation of the method. The specification also dictates to the developer of the method what behavior must be provided and the developer must meet the specification.

```
 /* precondition: s must contain a character array,
  *    delimited by the null character
  * postcondition: returns the length of s as an
  *    integer;
  */

int strlen( String[] s ) {

   // Implementation is irrelevant
   return(length);
}
```

Abstraction by Specification

The specification is a contract between the client and the class.

It tells the client what can be relied upon when calling the function or method. The client should not assume anything about the behavior or implementation of the method. The specification also dictates to the developer of the method what behavior must be provided and the developer must meet the specification.

The Public Interface of the class

Consider this…

```
{

    List list1 = new ArrayList();
    List list2 = new LinkedList();

}
```

`ArrayList` and `LinkedList` class both implement the List Interface, …

Consider this…

```
{

   List list1 = new ArrayList();
   List list2 = new LinkedList();

}
```

… and can be bound to the *behavior* of that type!

Consider this…
List is an abstraction (ADT)

```
{

   List list1 = new ArrayList();
   List list2 = new LinkedList();

}
```

*"… Data abstractions allow us to abstract from the way data structures are implemented to the **behavior** they provide that other programs can rely on…"*

Barbara Liskov

Abstract Data Types:

a summary

- An *abstract data type* (ADT) is a *logical* description of how we view some entity and the operations that are allowed to be performed on that entity.

Abstract Data Types:

a summary

- An *abstract data type* (ADT) is a *logical* description of how we view some entity and the operations that are allowed to be performed on that entity.
- It is a way of *classifying* a data type based on how objects of that type will be used and the behaviors they provide.
- The ADT does not specify how the data type must be implemented but simply provides *a minimal expected interface* and set of behaviors.

Abstract Data Types:

a summary

- An *abstract data type* (ADT) is a *logical* description of how we view some entity and the operations that are allowed to be performed on that entity.
- It is a way of *classifying* a data type based on how objects of that type will be used and the behaviors they provide.
- The ADT does not specify how the data type must be implemented but simply provides *a minimal expected interface* and set of behaviors.
- Allows us to focus on what the data type is representing and not with how it will eventually be constructed.
- Interfaces are **one way** we can specify an ADT in Java.
- Implementing the interfaces allows to create different implementations of the same ADT.

The List ADT:

another example

- A list is a sequence in which items can be accessed, inserted, and removed *at any position in the sequence*.
- The operations supported by our List ADT:
  - `getItem(i)`: get the item at position i
  - `addItem(item, i)`: add the specified item at position i
  - `removeItem(i)`: remove the item at position i
  - `length()`: get the number of items in the list
  - `isFull()`: test if the list already has the maximum number of items
- Note that we *don't* specify *how* the list will be implemented.

Specifying the List ADT Using an Interface

- In Java, we can use an interface to specify an ADT:

```
public interface List {
    Object getItem(int i);
    boolean addItem(Object item, int i);
    Object removeItem(int i);
    int length();
    boolean isFull();
}
```

- Again, the interface specifies a set of methods.
  - includes only their headers
  - does *not* typically include the full method definitions

Class vs. Type

a side note

- An object's *class* defines *how* the methods of an object are implemented, and it defines the internal state of an object.
- An object's type refers to an interface – the set of requests to which an object can respond.

The data members of the object.

Class vs. Type

a side note

- An object's class defines *how* the methods of an object are implemented, and it defines the internal state of an object.
- An object's *type* refers to an *interface* – the set of requests to which an object can respond.

Class vs. Type

a side note

- An object's class defines *how* the methods of an object are implemented, and it defines the internal state of an object.
- An object's *type* refers to an *interface* – the set of requests to which an object can respond.

This is **not** referring to a Java Interface. *The behaviors of the class themselves represent an interface. Java Interfaces are a language specific implementation of how to enforces a class's behavior and establish a* ***Type****.*

Class vs. Type

a side note

- An object's class defines *how* the methods of an object are implemented, and it defines the internal state of an object.
- An object's type refers to an interface – the set of requests to which an object can respond.
- An object can have many types, i.e.
  - polymorphic behavior
- Objects of different classes can have the same type, i.e.
  - multiple classes implementing the same behavior or interface.

Class vs. Type

- An object's class define
  implemented, and it de
- An object's type refers
  which an object can res

Given a student hierarchy, object `f` can be an instance of:

- *`Freshman`*
- `Undergraduate`
- *`Student … Comparable, etc.`*

- An object can have many types, i.e.
  - polymorphic behavior.
- Objects of different classes can have the same type, i.e.
  - multiple classes implementing the same behavior or interface.

Class vs. Type

- An object's class define
  implemented, and it de
- An object's type refers
  which an object can res

Given a student hierarchy, object `f` can be an instance of:

- *`Freshman`*
- `Undergraduate`
- *`Student`* … *`Comparable`*, *`etc.`*

- An object can have many types, i.e.
  - polymorphic behavior.
- Objects of different classes can have the same type, i.e.
  - multiple classes implementing the same behavior or interface.

Class vs. Type

a side note

- An object's class defines *how* the methods of an object are implemented, and it defines the internal state of an object.
- An object's type refers to an interface – the set of requests to which an object can respond.
- An object can have many types, i.e.
  - polymorphic behavior
- Objects of different classes can have the same type, i.e.
  - multiple classes implementing the same behavior or interface.

Objects of Shape and Animal can be drawable, comparable, etc.

First Principle of Good Design as stated in:

Elements of Reusable Object Oriented Software

- Program to an Interface and not an Implementation:
  - Do not declare variables to be an instance of particular concrete classes. Instead commit only to an interface as defined by an Abstract Class *or a Java Interface*.

First Principle of Good Design as stated in:

Elements of Reusable Object Oriented Software

- Program to an Interface and not an Implementation:
  - Do not declare variables to be an instance of particular concrete classes. Instead commit only to an interface as defined by an Abstract Class *or a Java Interface*.

First Principle of Good Design as stated in:

Elements of Reusable Object Oriented Software

- Program to an Interface and not an Implementation:
  - Do not declare variables to be an instance of particular concrete classes. Instead commit only to an interface as defined by an Abstract Class *or a Java Interface*.
  - Example:

```
public static void someMethod( List list ) {



}
```

First Principle of Good Design as stated in:

Elements of Reusable Object Oriented Software

- Program to an Interface and not an Implementation:
  - Do not declare variables to be an instance of particular concrete classes. Instead commit only to an interface as defined by an Abstract Class *or a Java Interface*.
  - Example:

```
public static void someMethod( List list ) {

// committing the parameter to type List ensures
that this method can be passed any object whose
class is of type List. In Java, any class that
implements the List interface.


}
```

First Principle of Good Design as stated in:

Elements of Reusable Object Oriented Software

- Program to an Interface and not an Implementation:
  - Do not declare variables to be an instance of particular concrete classes. Instead commit only to an interface as defined by an Abstract Class *or a Java Interface*.
  - Why?

First Principle of Good Design as stated in:

Elements of Reusable Object Oriented Software

- Program to an Interface and not an Implementation:
  - Do not declare variables to be an instance of particular concrete classes. Instead commit only to an interface as defined by an Abstract Class *or a Java Interface*.

1. Clients *remain unaware of the specific types of objects they use*, as long as the objects adhere to the interface that the clients expect.
2. Clients remain unaware of the classes that implement these objects. Clients are only aware of the type (abstract class or interface) that defines the object type interface.

First Principle of Good Design as stated in:

Elements of Reusable Object Oriented Software

- Program to an Interface and not an Implementation:
  - Do not declare variables to be an instance of particular concrete classes. Instead commit only to an interface as defined by an Abstract Class *or a Java Interface*.

1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that the clients expect.
2. Clients *remain unaware of the classes that implement these objects*. Clients are only aware of the type (abstract class or interface) that defines the object type interface.

Abstraction by Parameterization

*Abstraction by Parameterization seeks generality by allowing the same **function** to be adapted to many different contexts by providing it with the varying information on that context in a form of parameters.*

We do not write code that works on specific values, we write functions. Functions describe a computation that works on all acceptable values of the appropriate types. We specify those values in the form of parameters. Thus, the detail of what specific values are to be used is removed.

Parameterized types (e.g., Generics) are another example of abstraction by parameterization, where we vary the type of the parameter and not just the value passed.

Abstraction by Parameterization

Abstraction by Parameterization seeks generality by allowing the same function to be adapted to many different contexts by providing it with the varying information on that context in a form of parameters.

We do not write code that works on specific values, we write functions. Functions describe a computation that works on all acceptable values of the appropriate types. We specify those values in the form of parameters. Thus, the detail of what specific values are to be used is removed.

Parameterized types (e.g., Generics) are another example of abstraction by parameterization, where we vary the type of the parameter and not just the value passed.

Abstraction by Parameterization

Abstraction by Parameterization seeks generality by allowing the same function to be adapted to many different contexts by providing it with the varying information on that context in a form of parameters.

We do not write code that works on specific values, we write functions. Functions describe a computation that works on all acceptable values of the appropriate types. We specify those values in the form of parameters. Thus, the detail of what specific values are to be used is removed.

*Parameterized types (e.g., **Generics**) are another example of abstraction by parameterization, where we vary the type of the parameter and not just the value passed.*

Principle of Abstraction

- *Abstraction by Parameterization*

- *Abstraction by Specification*
  - *Modifiability*
  - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - *procedural*
  - *data*
  - *iteration*

Principle of Abstraction

> *… refers to the degree of ease that changes can be made to a software system, and the flexibility with which that software system can adapt to change in requirements.*

- *Abstraction by Parameterization*
- *Abstraction by Specification*
    - *Modifiability*
    - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
    - *procedural*
    - *data*
    - *iteration*

Principle of Abstraction

- *Abstraction by Parameteriza*
- *Abstraction by Specificati*
  - *Modifiability*
  - *Locality*

Locality allows a program to be implemented, understood, or modified one module or one class at a time without concern of breakage occurring elsewhere in the system.

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - *procedural*
  - *data*
  - *iteration*

Principle of Abstraction

- *Abstraction by Parameterization*

- *Abstraction by Specification*
  - *Modifiability*
  - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - *procedural*
  - *data*
  - *iteration*

Principle of Abstraction

- *Abstraction by Parameterization*

- *Abstraction by Specification*
  - *Modifiability*
  - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - *procedural*
  - *data*
  - *iteration*

Principle of Abstraction

- *Abstraction by Parameterization*

- *Abstraction by Specification*
  - *Modifiability*
  - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - ***procedural*** *// methods, parameters and returns*
  - *data*
  - *iteration*

Principle of Abstraction

- *Abstraction by Parameterization*

- *Abstraction by Specification*
  - *Modifiability*
  - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - *procedural* *// methods, parameters and returns*
  - ***data*** *// classes, inheritance, interfaces*
  - *iteration*

Principle of Abstraction

- *Abstraction by Parameterization*

- *Abstraction by Specification*
  - *Modifiability*
  - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - *procedural* *// methods, parameters and returns*
  - *data* *// classes, inheritance, interfaces*
  - ***Iteration*** *// collections*

Principle of Abstraction

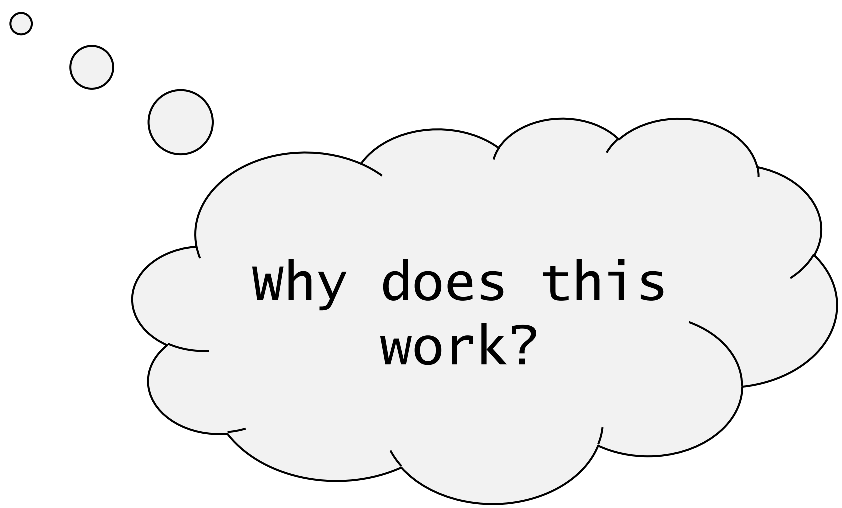
- *Abstraction by Parameterization*

- *Abstraction by Specification*
  - *Modifiability*
  - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - *procedural* *// methods, parameters and returns*
  - *data* *// classes, inheritance, interfaces*
  - *Iteration* *// collections*
  - ***type*** *// generics*

Consider this…

```
{

   List list1 = new ArrayList();
   List list2 = new LinkedList();

   list1.add( new Integer(3) );
   list1.add( new Student() );
   list1.add( new String("some string") );


}
```



Consider this…

```
{

   List list1 = new ArrayList();
   List list2 = new LinkedList();

   list1.add( new Integer(3) );
   list1.add( new Student() );
   list1.add( new String("some string") );

   ? item = list1.get(..);
}
```

What is the
type of the
object being
returned?

Consi

unbounded

```
{

    List list1 = new ArrayList();
    List list2 = new LinkedList();

    list1.add( new Integer(3) );
    list1.add( new Student() );
    list1.add( new String("some string") );

    ? item = list1.get(..);
}
```

What is the type of the object being returned?

Consider this…

```
{

    List<Object> list1 = new ArrayList();
    List list2 = new LinkedList();

    list1.add( new Integer(3) );
    list1.add( new Student() );
    list1.add( new String("some string") );

    ? item = list1.get(..);
}
```

Consider this…

```
{

    List<Object> list1 = new ArrayList();
    List list2 = new LinkedList();

    list1.add( new Integer(3) );
    list1.add( new Student() );
    list1.add( new String("some string") );

    Object item = list1.get(..);
}
```

Consider this…

```
{

   List<Object> list1 = new ArrayList();
   List list2 = new LinkedList();

   list1.add( new Integer(3) );
   list1.add( new Student() );
   list1.add( new String("some string") );

   Student item = list1.get(..);
}
```

Consider this…

```
{

    List<Object> list1 = new ArrayList();
    List list2 = new LinkedList();

    list1.add( new Integer(3) );
    list1.add( new Student() );
    list1.add( new String("some string") );

    Student item = list1.get(..);
}
```

Consider this…

```
{

   List<Object> list1 = new ArrayList();
   List list2 = new LinkedList();

   list1.add( new Integer(3) );
   list1.add( new Student() );
   list1.add( new String("some string") );

   Student item = (Student) list1.get(..);
}
```

explicit cast

Consi

bounded

```
{
    List<Student> list1 = new ArrayList<Student>();
    List list2 = new LinkedList();

    // list1.add( new Integer(3) );
    list1.add( new Student() );
    // list1.add( new String("some string") );

    Student item = list1.get(..);
}
```

Java uses **type erasure** during compilation and removes all type parameters and replaces it with the base type (*if bound*) or with Object (*if unbounded*).

Consi

Parameterize
the type!

```
{
    List<Student> list1 = new ArrayList<Student>();
    List list2 = new LinkedList();

    // list1.add( new Integer(3) );
    list1.add( new Student() );
    // list1.add( new String("some string") );

    Student item = list1.get(..);
}
```

Java uses **type erasure** during compilation and removes all type parameters and replaces it with the base type (*if bound*) or with Object (*if unbounded*).

Consi

Java Generics

```
{
    List<Student> list1 = new ArrayList<Student>();
    List list2 = new LinkedList();

    // list1.add( new Integer(3) );
    list1.add( new Student() );
    // list1.add( new String("some string") );

    Student item = list1.get(..);
}
```

Java uses **type erasure** during compilation and removes all type parameters and replaces it with the base type (*if bound*) or with Object (*if unbounded*).