To abstract is to eliminate distinctions

- Two things which are different at one level are similar or equivalent at a more abstract level.
- Upper classes are more abstract; lower classes are more concrete.
- The more concrete "subclass" extends the more abstract "superclass" and inherits the functionality of the superclass.
- The abstract class generalizes or abstracts the concrete class.
Maze Example

- A *Portable* is something that the explorer can pick up, carry, and put down.
- A *PortableList* describes the collection of items the explorer is carrying.
Extension and inheritance

- **Class extension**: the mechanism provided by the language for implementing abstraction.
- **Inheritance**: a mechanism by which a subclass automatically possesses all of the non-private features of its superclass.
public class ClosedFigure{
    public Location getLocation ()
    {
        return this.location;
    }
    public void setLocation (Location newLocation)
    {
        this.location = newLocation;
    }
    private Location location;
}
public class Circle extends ClosedFigure {

    public int radius (){...

    }

    Circle c = new Circle(10);
    Location l = c.getLocation();
    int d = c.radius();
A subclass does not “inherit” private features of its superclass.

The variable `location` is private to `ClosedFigure`.

Illegal:
```
this.location = null;
```

Legal:
```
this.setLocation(null);
```
Generalization and subtyping

- A reference to a subclass is also considered a reference to its superclass. i.e. a reference to a Circle can be considered a reference to a ClosedFigure.

- The type `reference-to-Circle` is a subset of the type `reference-to-ClosedFigure`.
Circle c = new Circle(10);
ClosedFigure f;
f = c;

BUT...
A ClosedFigure cannot be treated as a Circle. (f was a ClosedFigure, c was a Circle)

```java
int d;
Location l;
l = f.getLocation();  // legal
d = f.radius();        // Illegal
c = f;                 // Illegal
```
Suppose we want to get subclass information from \textit{f}.

First check to make sure it is the correct descendant and then cast the object \textit{f} to the inheriting class.

\begin{verbatim}
if (f instanceof Circle) {
    Circle c = (Circle) f;
    d = c.radius();
    ...
}
\end{verbatim}

\textit{subtyping}: a mechanism by which an instance of a subclass can be used in any context specifying the superclass.
Abstract class

- A class that is not intended to be instantiated should be declared as `abstract`.
- Instances of an abstract class cannot be created.
- It serves as a foundation on which to build subclasses.
- Non-abstract classes are called `concrete`. 
Abstract classes (cont.)

```java
public abstract class ClosedFigure {

    // We cannot instantiate an abstract class, but we can define variables that reference the class.
    ClosedFigure f;
    f = new ClosedFigure();  // Illegal
    f = new Circle(10);  // Legal.
```

![Class diagram](image.png)
Abstract methods

- Abstract classes may have *abstract methods*. An *abstract method* is specified, but not implemented.
- Since an abstract method cannot be implemented in the abstract class, any concrete subclass must provide an implementation for any abstract method it inherits.

```java
public abstract class ClosedFigure {
    public abstract int area();
    ...
}
```
Abstract methods (cont.)

```java
int d = f.area(); //Legal.
```

- If `f` references a `Circle` when the statement is executed, `f.area()` is directed to a `Circle`.
- If `f` references a rectangle when the statement is executed, `f.area()` is directed to a `Rectangle`.
- This mechanism is called *polymorphism*, *dynamic binding*, or *dispatching*.
Constructors

- Constructors (other than default) are not inherited.
- The constructor is responsible for initializing components defined in a superclass as well as those expressly defined in the class itself.
- A superclass constructor can be called: `super(arguments);`
public class Circle extends ClosedFigure {
    public Circle (int radius) {
        super();
        myRadius = radius;
    }

    public Circle () {
        this(1);   //invoke above constructor
    }

    ...

    private int myRadius;
}
Constructors (cont.)

- If no constructor is called for a superclass, and no constructor of the same class is called, the constructor is assumed to begin with the invocation of the superclass constructor super();

```java
public Circle (int radius) {
    super();     //assumed
    myRadius = radius;
}
```
Constructors (cont.)

- If a class doesn’t contain a constructor definition, a default constructor requiring no arguments is provided automatically.
  ```java
  public Class () {
    super();
  }
  ```

- The key words `this` and `super` as constructor invocations can appear only as the first statement of another constructor.
Overriding and polymorphism

- Every Java class comes equipped with a method `equals(Object obj)`.
- This is because every class is a subclass of `Object`.
- A class can redefine a method that it inherits.
- **Overriding**: providing an alternative implementation of an inherited method.
Overriding and polymorphism (cont.)

- The redefined method must have the same return type and the same number and type(s) of parameters as the inherited method.

```java
public class Circle extends ClosedFigure {

    ...

    public boolean equals (Object c) {
        if (c instanceof Circle)
            return this.radius() == ((Circle)c).radius();
        else
            return false;
    }

}```
ClosedFigure f1;
ClosedFigure f2;

int n;
// n is given a value here
if (n==0) {
    f1 = new Circle();
    f2 = new Circle();
} else {
    f1 = new Rectangle();
    f2 = new Rectangle();
}

boolean b = f1.equals(f2);
Overriding and polymorphism (cont.)

- **polymorphism**: dynamic behavior by which the method performed as the result of a call to a given object is determined at run-time by the class of the object.

- A class that overrides a method can call its parent’s overridden method.

```java
public boolean equals (Object c) {
    if (c instanceof ColoredCircle)
        return super.equals(c) &&
                this.colored().equals(c.color());
    else
        return false;
}
```
Method overloading

- A class can contain distinct methods with the same name as long as these methods differ in number, order, or type(s) of parameters.
- This is called **overloading**.

```java
public int m (int x);
public int m (Object obj);
public void m (int x, int y);
public void m (Object obj, int x);
public void m (int x, Object obj);

public Object m (int i);  //Illegal
```
Method overloading (cont.)

```java
public class Circle extends ClosedFigure {
    ...
    public boolean equals (Object obj) {
        ...
    }
    public boolean equals (Circle c) {
        return this.radius() == c.radius();
    }
}
```

- Overloading inherited methods can lead to confusion:

```java
ClosedFigure f = new Circle ();
Circle c = new Circle ();
boolean b1 = f.equals(c); // c is type Circle
boolean b2 = c.equals(f); // f is ClosedFigure
boolean b3 = c.equals((Circle)f); // (Circle) f // is a Circle
```
Method overloading (cont.)

- *Overloaded* methods are different methods in the same class.
- *Overriding* involves providing different implementations for a method in different classes.
Subclasses and contracts

- When overriding a method, postconditions can be strengthened but cannot be weakened.

```java
public void fill (Color c)
    Paint this ClosedFigure with the specified Color

    require:
        c != null

    ensure:
        this.backgroundColor().equals(c)
```
Subclasses and contracts (cont.)

```java
public class MonochromeFigure extends ClosedFigure {

    ...  

    /**
     * Paint this MonochromeFigure black or white
     * require:
     *  c.equals(Color.WHITE) ||
     *  c.equals(Color.BLACK)
     */

    public void fill (Color c) {
        Require.condition(
            c.equals(Color.WHITE) ||
            c.equals(Color.BLACK));
        ...
    }
}
```
Identifying common functionality

- The reason that we group classes together is because they have common attributes or functions.
- Suppose all the items that a maze game explorer can carry have \texttt{weight}; make \texttt{weight} a feature of the class \textit{Portable}.

\begin{verbatim}
public abstract class Portable {
    ...

    public int weight ()
    ...

}
\end{verbatim}
Identifying common functionality
(cont.)

```java
public class Pack {
    ...
    public int totalweight () {
        int sum = 0;
        int i = 0;
        while (i < items.size()) {
            sum = sum +
                items.get(i).weight();
            i = i +1;
        }
        return sum;
    }
    ...
    private PortableList items;
}
Alternative implementation for a method (cont.)

- We will extend the class `Player` from the nim game to include a `PerfectPlayer` that wins whenever possible.
- `PerfectPlayer` overrides the `makeMove` method.
- The creation of a new subclass does not require changes in a `Player` client.
Refining the implementation

- We override the method so that the subclass performs some action in addition to what is done in the superclass.
- The subclass, in overriding the method, also invokes the superclass method in the overriding definition.

```java
public class Wizard extends Explorer {
    ...
    public void addToPack (Portable item) {
        item.removeCurses();
        super.addToPack(item);
    }
    ...
}
```
Refining the implementation (cont.)

- General format:

  do something before
call the superclass method
do something after
Extending functionality

- Inheritance is used commonly to extend the functionality of a class.
- Example: We could extend the Player class of nim to include keeping score (the number of games Player has won).
- Sometimes additional functionality involves overriding existing methods.
Extending functionality (cont.)

- Example: Define a nim `Player` whose move is set by a client:

```java
class dummyPlayer extends Player {
    /**
     * Set number of sticks taken
     */
    public void setNumber (int n) ...
    /**
     * Take number of sticks specified
     */
    public void makeMove (Pile pile, int maximum) ...
}
```
So far, the methods defined in a class have been either public or private.

Features can also be protected or restricted.

Feature access has to do with the structure of the program text and not with the run-time state of the system; “from where in the program text can this feature be accessed?”
Feature accessibility (cont.)

```java
public class Circle {
    ...
    public boolean equals (Object obj) {
        if (obj instanceof Circle) {
            return myRadius ==
            ((Circle)obj).myRadius;
        } //legal functionality
        else
            return false;
    } //legal functionality
    ...
    private int myRadius;
}
public class ColoredCircle extends Circle{

    ...

    public boolean equals (Object obj) { 
        if (obj instanceof ColoredCircle){
            return myRadius ==
            ((Circle)obj).myRadius &&
            myColor ==
            ((ColoredCircle)obj).myColor;
        else
            return false;
        } /* illegal functionality; neither of the
        references to myRadius is legal */

    ...

    private int myColor;
}
public boolean equals (Object obj) {
    if (obj instanceof ColoredCircle) {
        return this.radius() ==
            ((Circle)obj).radius() &&
            myColor ==
            ((ColoredCircle)obj).myColor;
    } else
        return false;
} //LEGAL FUNCTIONALITY
Feature accessibility limited by class accessibility

- A class that is not public can be accessed only from within its own package.
- If a class is not visible outside its package, then neither are its features.
- Even if features are labeled *public*, access to features of a non-*public* class is limited to the package containing the class definition.
Protected features

- Private features of a class are not directly accessible in a subclass.
- A class inherits a feature specified as protected, and the feature is accessible in the class.
- The class Player has a component numberTaken.
- If this component is labeled protected, then subclasses like PerfectPlayer can access the variable directly.

```java
protected int numberTaken;
```
/**
 * This example is in the context of
 * numberTaken being private.
 */

public void makeMove (Pile pile,
                 int maximum) {

    int size = pile.size();
    int number;
    if (numberToTake <= size &&
        numberToTake <= maximum)
        number = numberToTake;
    else if (size <= maximum;
             number = size;
    else
        number = maximum
    pile.remove(number);

    this.numberTaken = number;//illegal access
}                      //numberTaken is private
Protected features (cont.)

- One approach:
  ```java
  protected int numberTaken;
  ```

- Better approach:
  ```java
  protected void setNumberTaken (int number)
  {
    numberTaken = number;
  }
  ```

  ```java
  private int numberTaken;
  ```
The accessibility of a protected feature is rather broad.

Class $BClass$ or $CClass$ can directly access any protected variables in $AClass$. 
Any class that inherits from or is in the same package as a class has access to the protected variables.
A better approach is to make the component private, and allow controlled access through protected methods.
Restricted features

- If a feature is declared neither `public`, `protected`, nor `private`, it is considered `restricted` (sometimes called “package”).
- There is no keyword `restricted`.
- Any class in the same package has access to restricted variables.
- If a feature is neither `public` nor `private`, the maintenance programmer must determine why this is so, and must assure that any use of the feature does not compromise the system.
Java scoping rules

- An identifier may occur many times in a program in many different contexts.
- An identifier can be introduced as the name of a variable, method, or parameter in a variable definition, method definition, or parameter specification.

```java
public void takeHit (int hitStrength)
private int tally;
int temp;
```
Java scoping rules (cont.)

- A defining occurrence establishes the identifier as the name of some entity.
- An applied occurrence is the use of an identifier to refer to the thing it names.
  \[ \text{tally} = \text{tally} + 1; \]
- An applied occurrence sometimes is prefixed with an object reference.
  \[ \text{this.room} = \text{monster.location}(); \]
Java scoping rules (cont.)

- **Scoping rules** are language rules that associate applied identifier occurrences with defining occurrences.

```java
public void increment () {
    tally = tally +1;
}
...
private int tally;
...
Java scoping rules (cont.)

- The scope of a local variable definition is from the definition to the end of the compound statement ({} or method body containing the definition.
- A parameter is treated like a local variable defined just inside the method body.

```java
public boolean smallerThan (Rectangle r) {
    double myArea;
    double yourArea;
    myArea = PI*radius*radius;
    yourArea = r.length() * r.width();
    return myArea < yourArea;
}
```
Java scoping rules (cont.)

```java
public class C {
    // can’t reference x or y here
    public void m(int x) {
        // can reference x here, but
        // can’t reference y
        int y;
        // can reference both
        // x and y here
    }
    // can’t reference x or y here
}
```
Java scoping rules (cont.)

```java
public boolean smallerThan (Rectangle R){
    double area1 = 100.0;
    double area2 = area1; // in scope of area1
}

public boolean smallerThan (Rectangle R){
    double area1 = area2; // NOT in scope
    double area2 = 100.0;
}
```
Java scoping rules (cont.)

A local variable definition can appear almost anywhere in a method body. The scope extends to the end of the compound statement containing the definition.

```java
public void m (int x) {
    if (x > 0) {
        int i = 1;
        ...
    }
    ...
}
```

`scope of int i`
Java scoping rules (cont.)

```java
public double m (int x) {
    double result;
    if (x > 1) {
        int sum = x;
        result = sum/2;  // scope of int sum
    }
    else {
        double sum = x;
        result = sum/2;  // scope of double sum
    }
    return result;
}
```

But this is not legal, since the scopes of the `int sum` and the `double sum` overlap:

```java
public double m (int x) {
    double result;
    double sum = x;
    if (x > 1) {
        int sum = x;  // scope of int sum
        result = sum/2;
    }
    else {
        result = sum/2;
    }
    return result;
}
```
An automatic variable can “hide” a component variable with the same name.

```java
public class Circle {
    private int radius;

    public void setRadius (int radius) {  
        this.radius = radius;
        //this.radius-- component variable
        //radius -- automatic variable
    }
}
```
We’ve covered

- Fundamental methods of modeling with objects: abstraction via class extension.
  - subclasses
  - extending functionality
- Subtyping
- Polymorphism
- Public, private, protected, and restricted features.
- Java scoping rules.