**Inheritance**

- Inheritance is a fundamental technique used to create and organize reusable classes.
- The child is a more specific version of the parent.
- The child inherits characteristics of the parent (methods and data defined by the parent class).
- Tailor derived class by adding new variables or methods, or by modifying the inherited ones.
- The keyword \texttt{extends} is used to establish an is-a (inheritance) relationship.

```java
class Child extends Parent {
    // class contents
}
```

**Class Hierarchies**

- A child class can be the parent of another child, forming a \textit{class hierarchy}.
- Two children of the same parent are called \textit{siblings}.
- Common features should be put as high in the hierarchy as is reasonable.
- An inherited member is passed continually down the line.
  - Therefore, a child class inherits from all its ancestor classes.

**Words.java**

```java
/**
 * Demonstrates the use of an inherited method.
 * @author Java Foundations
 */
public class Words {
    /**
     * Driver: Instantiates a derived class and invokes its inherited
     * and local methods.
     */
    public static void main(String[] args) {
        Dictionary webster = new Dictionary();
        System.out.println("Number of pages: " + webster.getPAGES());
        System.out.println("Number of definitions: " + webster.getDefinitions());
        System.out.println("Definitions per page: " + webster.computeRatio());
    }
}
```
The **protected** Modifier

- A **protected** variable is visible to any class in the same package as the parent class.

- The **protected** modifier allows a child class to reference a variable or method directly in the parent class.

- It provides more **encapsulation** than **public** visibility, but is not as tightly encapsulated as **private** visibility.

The **super** Reference

- Constructors are **not** inherited, even though they have public visibility.

- Yet, we often want to use the parent's constructor to set up the "parent's part" of the object.

- The keyword **super** can be used to refer to the parent class, including the parent's constructor.

- A child's constructor should:
  - Call the parent's constructor as its first line: **super();**
  - If it does not call **super()**, a 0-parameters **super()** constructor will be called anyway!

- The **super** reference can also be used to reference other variables & methods defined in parent's class.
HungerGamesDictionary

A HungerGamesDictionary object keeps track of definitions related to terms in the Hunger Games (such as the word "mockingjay") as well as the number of districts in Panem.

Write a HungerGamesDictionary class that inherits from Dictionary2 and contains the following:

- An integer instance variable that keeps track of the number of districts.
- A constructor with three parameters corresponding to the number of pages, the number of definitions, and the number of districts.
- A second constructor with two parameters corresponding to the number of pages, the number of definitions. The constructor initializes the number of district to 13.
- A toString method that returns a tab-delimited String consisting of the number of pages, the number of definitions, and the the number of districts.
A child class can override the definition of an inherited method in favor of its own.

A method in the parent class can be invoked explicitly using the super reference, as in:

```java
super.message();
```

If a method is declared with the final modifier, it cannot be overridden.

The concept of overriding can be applied to variables and is called shadowing variables.

Shadowing variables should be avoided because it tends to cause unnecessarily confusing code.
Overloading vs. Overriding

- **Overloading** deals with multiple methods with the same name in the same class, but with different signatures.
- **Overriding** deals with two methods, one in a parent class and one in a child class, that have the same signature.
- Overloading lets you define a similar operation in different ways for different parameters.
- Overriding lets you define a similar operation in different ways for different object types.

Class Hierarchies

- A child class can be the parent of another child, forming a class hierarchy.
- Two children of the same parent are called siblings.
- Common features should be put as high in the hierarchy as is reasonable.
- An inherited member is passed continually down the line.
  - Therefore, a child class inherits from all its ancestor classes.

An Alternative Class Hierarchy

There is no single class hierarchy that is appropriate for all situations.

The Object Class

- A class called Object is defined in the java.lang package of the Java standard class library.
- All classes are derived from the Object class.
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the Object class.
- Therefore, the Object class is the ultimate root of all class hierarchies.
The Object Class Methods
- The Object class contains a few useful methods, which are inherited by all classes.
- I.e., the toString() method is defined in the Object class.
- Every time we define the toString method, we are actually overriding an inherited definition.
- The toString method in the Object class is defined to return a string that contains the name of the objects class along with some other information.
- Also in Object:
  - equals() returns T if and only if __________
  - clone() returns __________

Abstract Classes
- An abstract class is a placeholder in a class hierarchy that represents a generic concept.
- An abstract class cannot be instantiated.
- To declare a class as abstract:
  ```java
  public abstract class Shape
  {
    // contents
  }
  ```
- Abstract classes are an important element of software design: they allow us to establish common elements in a hierarchy that are too generic to instantiate.

Need for Abstract Classes

Abstract Classes: Rules
- An abstract class often contains abstract methods with no definitions:
  - The abstract modifier must be applied to each abstract method.
- An abstract class typically contains non-abstract methods with full definitions.
- A class declared as abstract does not have to contain abstract methods – simply declaring it as abstract makes it so.
- The child of an abstract class must override the abstract methods of the parent, or it, too, will be considered abstract.
- An abstract method cannot be defined as final or static.