Subclassing

A class X can be *extended* to form a new class Y.

Then X is called the *superclass* of Y, and Y is called a *subclass* of X.

A subclass inherits members from its superclass.

```
public class MyPRNGGlorified extends MyPRNG
{
  private int usageCount = 0;

  public MyPRNGGlorified() {
    super();
  }

  public MyPRNGGlorified(int seed) {
    super(seed);
  }

  public int glorified_gimme() {
    System.out.print("I strike ");
    System.out.print(++usageCount);
    System.out.println(" times!");
    return gimme();
  }
}
```


Constructor Chaining

Use `super` to call a constructor of the superclass.

Use `this` to call a constructor of my class.

You can only have one of them.

It must be at the beginning of your constructor.

```
public MyPRNGGlorified(int seed) {
  super(seed);
}

public MyPRNGGlorified(int seed, int startWith) {
  this(seed);
  usageCount = startWith;
}
```

If you use neither, an implicit call to `super()` will be inserted.

```
public MyPRNGGlorified(boolean b) {
  // implicitly calls super() first
  if (b) usageCount = 1;
}
```

Exercise: what if superclass has no default constructor?

Constructor Execution

The execution order of constructors and initializations:

1. set all fields, including those of its class and superclass and so on, to default values

2. execute constructor of my class:

   (a) call constructor of superclass
   (b) run initializers and initialization blocks of my class
   (c) run the rest of my constructor

Discuss: why is this the right way?

Constructor Execution Example

```
public class MyPRNG
{
  private int current;

  public MyPRNG(int seed) {
    current = seed;
  }
}

public class MyPRNGGlorified extends MyPRNG
{
  private int usageCount = 0;

  public MyPRNGGlorified(int seed, int startWith) {
    super(seed);
    usageCount = startWith;
  }
}
```

1. set `current` to 0, `usageCount` to 0 (default values)
2. do `MyPRNGGlorified()`
   (a) do `MyPRNG()`
      i. set `current` to `seed`
   (b) set `usageCount` to 0 (initializer)
   (c) set `usageCount` to `startWith`

Access Control

Recall that a member of a class could be declared *public* or *private*. It can also be declared *protected*:

public accessible to all classes

private accessible to the defining class

protected accessible to the defining class and its subclasses

```
public class MyPRNG
{
  protected int current;
  // ...
}

public class MyPRNGGlorified extends MyPRNG
{
  // ...
  public void foo() {
    System.out.println(current);
  }
}
```


Method Overriding

A method in the superclass is *overriden* by a method of the same parameter and return types in the subclass.

```
public class MyPRNG
{
  public int gimme() { /* ... */ }
}

public class MyPRNGGlorified extends MyPRNG
{
  public int gimme() { /* !!! */ }
}

// ...
MyPRNGGlorified r = new MyPRNGGlorified();
MyPRNG s = r;
r.gimme();
s.gimme();
```

Both calls to `gimme` invoke the same version—that of `MyPRNGGlorified`.

Rules of Method Overriding

The version in the subclass must:

- have the same parameter types and return type as the superclass version
- have the same or more relaxed access compared to the superclass version
- have in its `throws` clause no exception that the superclass version does not have

Field Hiding

A field in the superclass is *hidden* by a field of the same name in the subclass.

```
public class MyPRNG
{
  public int x = 0;
}

public class MyPRNGGlorified extends MyPRNG
{
  public double x = 1.1;
}

// ...
MyPRNGGlorified r = new MyPRNGGlorified();
MyPRNG s = r;
System.out.println(r.x);  // 1.1
System.out.println(s.x);  // 0
```

Which `x` you get depends on the type of the *reference* you use.

The `super` Reference

The reference `super` refers to the current object, but pretends that it is an instance of its superclass.

```
public class MyPRNG
{
  public int x = 0;
  public int gimme() { /* ... */ }
}

public class MyPRNGGlorified extends MyPRNG
{
  public double x = 1.1;
  public int gimme() { /* !!! */ }
  public void foo() {
    System.out.println(x);        // mine
    System.out.println(super.x);  // MyPRNG's
    gimme();                      // mine
    super.gimme();                // MyPRNG's
  }
}
```


Subclassing and Arrays

A reference to an array of a subclass can be assigned to a reference to an array of the superclass.

```
MyPRNGGlorified[] R = new MyPRNGGlorified[10];
// ...
MyPRNG[] S = R;
```

If you plan to hold some objects of the subclass and some objects of the superclass in the same array, you must use an array of the superclass.

```
MyPRNG[] T = new MyPRNG[2];
T[0] = new MyPRNG();
T[1] = new MyPRNGGlorified();
```


Destructor Chaining

The superclass may have a `finalize()` method for cleaning up its (possibly private) state, e.g., closing a file.

If the subclass also has `finalize()`, it should call the `finalize()` of the superclass so as not to miss that.

```
public class MyPRNGGlorified
{
  protected void finalize() throws Throwable {
    // ... some cleanup
    super.finalize();
    // ... more cleanup
  }
}
```

Make it a habit that, whenever you write a `finalize()` method, *always* call `super.finalize()` inside it.

Exercise: Why?

(Hint: what if the superclass is changed afterwards?)

The `Object` Class

The built-in `Object` class is the universal base class.

If your class does not have an `extends` clause, it is implicitly `extends Object`.

Thus all classes inherit all members of the `Object` class. Some of them are:

- `protected void finalize() throws Throwable`

  called by garbage collector

- `public String toString()`

  returns a string form of this object

- `public boolean equals(Object obj)`

  compares this and obj for equality

- `public int hashCode()`

  hashing function; returns a hash value of the object