Abstract Classes and Methods

Usually, some base class declares the basic operations relevant to itself and all subclasses.

- `Object` provides a `finalize` operation.
- `MyPRNG` provides a `gimme` operation.

These two also provide default behaviour for convenience.

But in some other cases, no default behaviour is sensible.

- a `Shape` base class with subclasses such as `Rectangle`, `Circle`, etc. We need an operation `area` computing their areas. This method should be declared in `Shape`, but probably you do not want a default behaviour for it.

Abstract Classes and Methods

To declare a method without giving an implementation, do these:

1. Declare the class `abstract`.
2. Declare the method `abstract`.

E.g., the `Shape` class:

```
public abstract class Shape
{
  public abstract double area();
  // other stuff
}
```

You can even do this to an inherited method.

```
public abstract FunnyPRNG extends MyPRNG
{
  public abstract int gimme();
}
```


Extending Abstract Classes

A subclass of an abstract class may implement an inherited abstract method:

```
public class Circle extends Shape
{
  protected double radius, centreX, centreY;
  // ...
  public double area() {
    return radius * radius * Math.PI;
  }
  // ...
}
```

A subclass may also leave an inherited abstract method unimplemented. Then the subclass is also an abstract class.

```
public abstract class Shape2 extends Shape
{
  // does not implement area()
}
```


Abstract Class vs. Concrete Class

A class with unimplemented abstract methods is an *abstract* class. You cannot create instances of it.

A class with all methods implemented is a *concrete* class. You can create instances of it.

```
new Shape();   // illegal
new Shape2();  // illegal
new Circle();  // ok
```

Interfaces

An *interface* in Java is an abstract class taken to the extreme.

An interface is like a class, except that all methods are public and abstract, and all fields are public, static, and final—i.e., essentially public constants.

```
public abstract interface Sequence
{
  public static final int ZERO = 0;
  public abstract int gimme();
}
```

Those keywords may be omitted, as there is no choice anyway.

```
public interface Sequence
{
  int X = 1;
  int gimme();
}
```


Implementing an Interface

A class may implement an interface. This is like extending a class—all members are inherited.

```
public class MyPRNG implements Sequence
{
  //...
  public int gimme() {
//...
```

Same rules regarding overriding, overloading, and leaving an inherited abstract method unimplemented.

An instance of the class can be used wherever an instance of the interface is expected. (Again like subclassing.)

```
Sequence s = new MyPRNG();
int x = s.gimme();
```


Interface vs. Superclass

Some differences between implementing an interface and extending a superclass:

- A superclass can provide protected members and default behaviour to aid subclasses, whereas an interface cannot.
- A subclass can extend only one superclass, but can implement several interfaces (along with extending from a superclass).

  ```
  public class B extends A implements X, Y
  ```

- New interfaces may extend one or more existing interfaces

  ```
  public interface Z extends X, Y
  ```


Type Determination

The `instanceof` operator determines during runtime if an object is of the type you want. Suppose we have:

```
Object x = new MyPRNGGlorified();
```

Then the following are true:

```
x instanceof Object
x instanceof MyPRNG
x instanceof MyPRNGGlorified
x instanceof Sequence
```

and `x instanceof` *type* is false for any other *type*.

The result only depends on the actual object, not the type of the reference pointing to it. Suppose we have instead:

```
MyPRNGGlorified x = new MyPRNGGlorified();
```

Then all of the above still hold.

Type Determination

General rule of the `instanceof` operator:

```
x instanceof t
```

is true iff:

- if `t` is a class, then it must be the same as or a superclass of the actual class of `x`

- if `t` is an interface, then it, or one of its extensions, must be implemented by the actual class of `x` or a superclass of it

The tests are performed during runtime.

Type Casting

If an object is of a type you want but the reference you have is not, you can use type casting to get a reference of the desired type. E.g.,

```
Object x = new MyPRNGGloried();
// x.gimme() does not make sense
MyPRNG y = (MyPRNG)x;
// now y.gimme() makes sense
int a = y.gimme();            // cool
int b = ((MyPRNG)x).gimme();  // also cool
```

The validity of type casting of objects is checked only during runtime. It is valid iff `x instanceof t` would be true. When it is invalid, an exception will occur.

```
((Shape)x).area();  // exception!
```

You may like to use `instanceof` to test for the validity yourself before casting.