Classes and Objects

A *class* can be thought of as a data type.

An *object* is an *instance* of its class or data type.

In a class we declare *fields*, *constructors*, and *methods*. They are collectively called *members*.

Fields State variables storing the state of the object.

Constructors Procedures for initializing the fields.

Methods Functions for manipulating the object, in particular its fields.

Each member can be declared *public* or *private*:

public accessible to all classes

private accessible to only the defining class

Class Example

```
public class MyPRNG
{
  private int current;

  public MyPRNG() {
    current = 10;
  }

  public int gimme() {
    next();
    return current;
  }

  private void next() {
    current = (current * 17) % 19;
  }
}
```


Class Example (cont.)

```
public class UsePRNG
{
  public static void main(String[] args) {
    MyPRNG g1 = new MyPRNG();
    MyPRNG g2 = new MyPRNG();
    MyPRNG g3 = g1;

    System.out.println(g1.gimme());
    System.out.println(g2.gimme());
    System.out.println(g3.gimme());
  }
}
```

Note that g1, g2, and g3 are only *references*. You must set them to actual objects like in the above.

(cf. arrays. In fact we can consider `int[]` a class name; objects of this class have an operation called `[]` and a field called `length`.)

Constructor Overloading

Sometimes we want choice when creating objects.

We want several constructors.

E.g., the *default* constructor for default initialization:

```
public MyPRNG() {
  current = 10;
}
```

and a constructor for user-provided seed:

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

You can now create `MyPRNG` objects like this:

```
MyPRNG g3 = new MyPRNG(11);
```

FAQ: What if I just write no constructor at all?

Answer: Then a default constructor with no code is inserted automatically.

Constructor Overloading (cont.)

The keyword `this` allows you to make calls between constructors. Handy for reusing code.

E.g., can rewrite the default constructor in terms of the other one:

```
public MyPRNG() {
  this(10);
}
```

But you can do it at most once in each constructor, and it must be the first statement if you do it.

Alternative style: always call the default constructor first, then customize some fields:

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

Useful when there are many fields to initialize.

Method Overloading

Sometimes we want to offer alternative forms of the same operation.

We want several methods of the same name but different parameter types.

E.g., an alternative of `gimme` that skips any number of intermediates:

```
public int gimme(int n) {
  for (int i = 0; i < n; i++) next();
  return current;
}
```

Use like this:

```
r = g1.gimme(10);
```


Class-Wide Members (Or, `static` Explained)

You can declare a field or method `static`.

The member is shared by all objects of the class.

The member exists even when no object of the class exists.

```
public static final int max = 18;
public static final int min = 0;
public static int range() {
  return max - min;
}
```

Use like this:

```
m = MyPRNG.max;
n = MyPRNG.min;
r = MyPRNG.range();
```

Or this:

```
m = g1.max;
n = g1.min;
r = g1.range();
```


Initialization Blocks

Sometimes you want to initialize a `static` field by a code fragment.

E.g., an array containing perfect squares: 1^2, ..., 100^2. Too tedious to type them out.

```
private static int[] squares = new int[100];

static {
  for (int i = 0; i < 100; i++)
    squares[i] = (i+1)*(i+1);
}
```

Similarly you can also initialize object fields outside constructors.

```
private int[] cubes = new int[100];

{
  for (int i = 0; i < 100; i++)
    squares[i] = (i+1)*(i+1)*(i+1);
}
```

Arrays of Objects

```
MyPRNG[] gs = new MyPRNG[10];
```

Naturally, this only gives you an array of *references*.

(Consistent with the semantics of a declaration like `MyPRNG g;`)

You must then set each element to an actual object, existing or new. (Otherwise you would get the `null` pointer.)

E.g.,

```
{
  for (int i = 0; i < 10; i++)
    gs[i] = new MyPRNG();
}
```

Then you can use the elements like this:

```
r = gs[3].gimme();
```


The `this` Reference

You can refer to the current object by the keyword `this` inside an object method.

E.g., could write the following, though silly:

```
public int gimme() {
  this.next();
  return this.current;
}
```

More useful in this fashion, though not a good idea:

```
public void reset(int current) {
  this.current = current;
}
```

Also important in passing "myself" to another method, e.g.,

```
// add myself to some hypothetical list
GlobalLinkedList.add(this);
```


Object Destruction

Java does garbage collection.

Orphaned objects will have their memory space reclaimed.

When an orphaned object is collected, its `finalize` method will be called if it has one.

```
protected void finalize() throws Throwable {
  // stuff
}
```

Imperative for objects that allocate non-memory resources.

E.g., if your object owns an opened file, you should close it in `finalize`. Probably you should also provide a method for closing it.